

**PRACTICAL MICROCOMPUTER  
PROGRAMMING:**

3983 222839 XEQ2 SHLD JMP+1  
3986 CDD83D CALL FPRM  
3989 28EB BZ XEQ0  
398B FEAE CPI  
398D 20EF BNE EQ1  
398F 224039 XEQ3 SHLD A  
14239 LXI BPI  
010300 LXI B,3  
EDB0 MW+R  
ED584039 LDED BPA  
214A39 LXI H,RTINST  
010300 LXI B,3  
A4 EDB0 MW+R  
9A6 3E01 MYI  
9A8 324939 STA ILOC  
39AB C31139 JMP JM2  
39AE ED4B4039 XEQ4 LBCD BPA  
39B2 78 MOV A,B  
39B3 FEFF CPI FF

MP ADDRESS  
BKPT ADDRESS  
IN PERIOD?  
GARBAGE  
AVE BKPT ADDRESS  
SAVE PROGRAM  
MOVE IN BKPT RN INST.  
CONTINUE PREVIOUS BKPT  
WHERE A PREVIOUS P

**THE  
Z80**

**PRACTICAL  
MICROCOMPUTER  
PROGRAMMING:  
THE Z80**

**WELLER**

Northern  
Technology  
Books

Including source listings of  
conversational assembler and debugging monitor

**W.J. WELLER**



# **PRACTICAL MICROCOMPUTER PROGRAMMING: THE Z80**

**W.J. WELLER**

This third volume of the PRACTICAL MICROCOMPUTER PROGRAMMING™ series is concerned with detailed assembly language programming procedures for the Z80 microcomputer. In 18 chapters and four appendices it supplies everything necessary to write and debug Z80 application programs, including an assembler and debugging monitor. Paper tape object copies of this software are supplied free to the purchaser of this book with the return of the coupon in the back.

The 18 chapters of text cover all of the fundamental assembly level programming techniques, reinforced by more than 100 formal, tested examples which illustrate the techniques being discussed.

## **About the Author**

**Walt Weller** is an application software consultant specializing in industrial, medical and educational uses of small computers. In addition to this book he is the author of **Practical Microcomputer Programming: The M6800 and Assembly Level Programming for Small Computers** (Lexington, 1975) and coauthor of **Practical Microcomputer Programming: The Intel 8080 and An Editor/Assembler System for 8080/8085 Based Computers**. He currently resides in the Chicago area.

---

Additional copies of this book are available prepaid from:

Northern Technology Books  
Box 62  
Evanston, IL 60204

---

### *Other titles in this series:*

|                                                        |         |
|--------------------------------------------------------|---------|
| Practical Microcomputer Programming:<br>The Intel 8080 | \$21.95 |
| Practical Microcomputer Programming:<br>The M6800      | \$21.95 |



**PRACTICAL MICROCOMPUTER  
PROGRAMMING: THE Z80**

**WELLER**

**PRACTICAL  
MICROCOMPUTER  
PROGRAMMING.**

---

**The Z80**

by  
**W. J. Weller**



# PRACTICAL MICROCOMPUTER PROGRAMMING:

---

## The Z80

by  
W. J. Weller



Northern Technology Books

Weller, Walter J.

Practical Microcomputer Programming: The Z80

Library of Congress Catalog Card #77-88863

International Standard Book Number: 0-930594-05-3

Copyright © Walter J. Weller 1978

All rights reserved. No part of this publication may be reproduced, copied or transmitted in any form or by any mechanical or electronic means without written permission of the copyright holder.

Second Printing May 1979



To Ruth Marie





# TABLE OF CONTENTS

|                  |                                                                                                                                                                                                                                                                                                                                                                                                                                                  |           |
|------------------|--------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------|-----------|
| <b>Chapter 1</b> | <b>The Nature of the Programming Task</b>                                                                                                                                                                                                                                                                                                                                                                                                        | <b>1</b>  |
|                  | The difficulty of programming.<br>Literal nature of machines.<br>Programming languages.                                                                                                                                                                                                                                                                                                                                                          |           |
| <b>Chapter 2</b> | <b>Binary Operations</b>                                                                                                                                                                                                                                                                                                                                                                                                                         | <b>6</b>  |
|                  | Two state representation of information.<br>Two state codes. Meaning of place values in decimal and binary. Powers of two. Octal-binary conversions. Hexadecimal-binary conversions. The Inclusive OR function. The AND function. The Exclusive OR function. Complementation. Shifting. Binary addition. Binary subtraction. Two's complement representation. The extended sign bit. Meaning of overflow and carry. The maximum negative number. |           |
| <b>Chapter 3</b> | <b>Logical Organization of the Z80 Computer</b>                                                                                                                                                                                                                                                                                                                                                                                                  | <b>27</b> |
|                  | Organization of memory into words. Word length. The flag register. Definition of a program. The CPU. The program counter. Instruction execution cycle. Z80 registers.                                                                                                                                                                                                                                                                            |           |
| <b>Chapter 4</b> | <b>Machine and Assembly Language Programming</b>                                                                                                                                                                                                                                                                                                                                                                                                 | <b>32</b> |
|                  | The binary program. Symbolic programming. The assembler. The source program. The assembly listing. Label, instruction and operand fields. The                                                                                                                                                                                                                                                                                                    |           |

comments field. Error flags. Rules for label information.

**Chapter 5      Using the Assembly Program      41**

The text editor. The source buffer. Adding and deleting text from the buffer. Listing and punching the buffer. Reading tape programs. Assembling the program.

**Chapter 6      Moving Data with Z80 Instructions      54**

Concepts of loading and storing. Direct loading and storing. Use of pointers to load and store. Use of indexed addressing and the displacement. Immediate addressing mode. Data types in immediate instructions. Moving blocks of data. Looping under control of a count. Use of **DBNZ**. Condition jumps and branches. Use of the Z80 block move instructions. A FIFO stack example.

**Chapter 7      Arithmetic and Logical Operations on the Z80      78**

Definitions of bits in the Z80 flags. Addition of single registers. Addition of memory to A through pointers. Addition including previous carry. Double precision addition. Subtraction with and without previous borrow. Double precision subtraction. Propagation of carry. Double precision addition and subtraction in register pairs. A quadruple precision addition. Scaling of fixed point numbers. Rotary, logical and arithmetic shifts. The Z80 shift instructions. Multiple precision shifts. Detection of overflow in arithmetic shifts. Z80 logical instructions.



|                   |                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                   |            |
|-------------------|-------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------|------------|
|                   | Comparison operations on signed and unsigned quantities. Table of conditional jumps and branches. Comparison between inclusive limits. Negation in multiple precision.                                                                                                                                                                                                                                                                                                                                                                                            |            |
| <b>Chapter 8</b>  | <b>Software Multiplication and Division</b>                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                       | <b>113</b> |
|                   | Multiplication by a constant multiplier. Unsigned 8 by 8 bit multiply. Signed 8 by 8 bit multiply. Unsigned 16 by 16 bit multiply. Signed 16 by 16 bit multiply. Scale factor of product. Constraints on division. The divide fault condition. Setting quotient and remainder signs. Double precision division. Approximation for division of fractions.                                                                                                                                                                                                          |            |
| <b>Chapter 9</b>  | <b>Using the Stack Pointer</b>                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                    | <b>135</b> |
|                   | Definition of the stack. Access to the stack by pushing and popping. Sequence of events in <b>PUSH</b> and <b>POP</b> . Use of stack for temporary storage. Register swaps using the stack. Use of the stack to access data. Accessing the top of the stack. Concept of a subroutine. Sequence of events during a <b>CALL</b> . Return from the subroutine. Subroutine arguments in Z80 registers. Arguments in line after <b>CALL</b> . Argument addresses in line after <b>CALL</b> . Conditional call and return instructions. Synonyms for use after compare. |            |
| <b>Chapter 10</b> | <b>Handling Arrays and Tables</b>                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                 | <b>162</b> |
|                   | Array space reservation by <b>RES</b> , <b>BSS</b> and <b>ASC</b> . A simple table search. Table search using the Z80 block compare instructions. Search of multilength entry table for                                                                                                                                                                                                                                                                                                                                                                           |            |

reference to another table. The principle of sorting. The “bubble” sort. Use of argument to compute position in table.

|                   |                                                                                                                                                                                                                                                                                                                                                                                                                           |            |
|-------------------|---------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------|------------|
| <b>Chapter 11</b> | <b>Decimal Arithmetic</b>                                                                                                                                                                                                                                                                                                                                                                                                 | <b>182</b> |
|                   | Definition of BCD. The decimal adjustment of the binary sum of two BCD numbers — the <b>DAA</b> instruction. Multiple precision decimal addition. Adjustment of the difference of two BCD numbers. Multiple precision decimal subtraction. BCD shifting with <b>RLD</b> and <b>RRD</b> . Decimal multiplication. Decimal division.                                                                                        |            |
| <b>Chapter 12</b> | <b>Communication with Terminals</b>                                                                                                                                                                                                                                                                                                                                                                                       | <b>196</b> |
|                   | Difference of speed between computer and peripheral. The peripheral interface. The device address. Control, sense, and data transfer operations. The control and status ports. Delay by countdown. Delay by ready flag test. String print by counting characters. String print by testing for special character. Keyboard read with echo. Complex input/output facilities on the Z80. ASCII code table.                   |            |
| <b>Chapter 13</b> | <b>Number Base Conversions</b>                                                                                                                                                                                                                                                                                                                                                                                            | <b>211</b> |
|                   | The numerical ASCII codes. Validity check on incoming data. Decimal to binary integer conversion. Double precision decimal to binary integer. Quadruple precision decimal to binary integer. Decimal fraction to binary fraction. Binary integer to decimal integer by division. Binary integer to decimal integer by subtraction. Elimination of leading zeros. Conversion of signed numbers. Binary fraction to decimal |            |

fraction by multiplication. Conversion of octal and hexadecimal.

**Chapter 14      Floating Point Arithmetic      235**

Parts of a floating point number. Exponent bias. Normalized and unnormalized numbers. Equalization of exponents for addition and subtraction. Floating overflow and underflow. The pre-addition shift. Dependence of precision on order of operations. The guard digit. Noisy mode arithmetic. The true hexadecimal exponent. Conversion of floating point numbers — the radix deflection method. Considerations of precision loss during conversion of floating point numbers.

**Chapter 15      Graphic Output      250**

A simple graphic display — the frequency display. Building the frequency table. Varying line length under control of a count. Concept of in-memory image. The image region. Construction and printing of a simple X-Y graph. Computation of address in the image region. The bit manipulation instructions. A more complex graph — the Hertzsprung-Russell diagram. The variable plotting character as a device to show the third dimension.

**Chapter 16      Programming with Interrupts      276**

The reason for interrupts. Sequence of events during an interrupt. Conditions required for an interrupt. The restart instruction. Z80 interrupt modes. The interrupt vector register. Vectored interrupts. The real time clock interrupt.



A software time-of-day clock. Delay of enablement after EI. Removal of the source of interrupt by data transmission. Special considerations in output interrupt service. Obligations of the interrupt service routine. An interrupt driven output routine. The nonmaskable interrupt. Return from nonmaskable interrupt. The problem of reentrancy and priority interrupt systems. Some common interrupt programming errors. In line use of **RST/TRP** instructions.

**Chapter 17      Program Debugging Techniques      297**

Introduction to the ZEDBUG debugging monitor. Loading the object program. Checksum errors. Examination and modification of memory. Displaying blocks of memory. Filling and searching blocks of memory. Input from and output to peripheral devices. The ZEDBUG pseudo registers. Display and modification of the pseudo registers. Setting the bits of pseudo F and F'. Concept of breakpointing. Breakpointing through an example program. Some common errors. Replacing an instruction. Some cautions about breakpointing. Removing an offending instruction. Patching in an extra instruction. Use of the **RST/TRP** instruction to patch. The jump command. Enablement/disablement of interrupts during breakpointing and jumping. Examination of stack contents with the M command.

**Chapter 18      Functional Description      317**  
**of the Assembly Program**

**Appendices**

|              |                                                            |            |
|--------------|------------------------------------------------------------|------------|
| <b>A</b>     | Procedure for loading ZEDBUG<br>and Assembler object tapes | <b>327</b> |
|              | Sample bootstrap loader                                    | <b>332</b> |
| <b>B</b>     | Assembler I/O driver specifications                        | <b>334</b> |
|              | Summary of Editor/Assembler commands                       | <b>339</b> |
|              | Sample I/O drivers for Assembler                           | <b>343</b> |
|              | Assembly listing of the Assembler                          | <b>348</b> |
| <b>C</b>     | ZEDBUG I/O driver specifications                           | <b>425</b> |
|              | Sample I/O drivers for ZEDBUG                              | <b>430</b> |
|              | Summary of ZEDBUG commands                                 | <b>433</b> |
|              | Assembly listing of the ZEDBUG monitor                     | <b>436</b> |
| <b>D</b>     | Cross index of Assembler mnemonics<br>with Zilog mnemonics | <b>455</b> |
|              | Summary of Assembler pseudo operations                     | <b>474</b> |
| <b>Index</b> |                                                            | <b>477</b> |

# PREFACE

This third volume of the Practical Microcomputer Programming™ series is concerned with detailed assembly language programming procedures for the Z80® microcomputer manufactured by Zilog, Inc. of Cupertino, California. Its purpose is to provide the reader with the necessary information and software tools to make effective use of the Z80. The software, an assembler and debugging monitor, is given in full source form in appendices, and the purchaser of this book will be granted license to make copies for his personal or academic, but not commercial, use when the coupon at the back of the book is fully filled out and returned. To save the tedium of retranscribing the programs, paper tape object copies of both assembler and debugging monitor will be sent to the purchaser at no further cost upon receipt of the filled out coupon.

From one viewpoint or another every computer design contains flaws or features which might be viewed as awkward. The Z80 is no exception to this. Its weak points notwithstanding, a detached observer must conclude that, from a programming point of view, the Z80 is the most powerful eight bit microcomputer yet to appear. Designed as a superset of its popular predecessor the 8080, it offers enhancements over the 8080 which remove almost all of the objections to the programming characteristics of the earlier machine. It has a flexible, powerful instruction set which allows it, in skilled hands, to perform tasks of any significant size and complexity in substantially less memory than competitive microcomputers. With a single minor exception, its binary instruction set is a superset of the 8080's, which allows software developed for the 8080 to run unchanged on the Z80, an important consideration when contemplating a machine upgrade.

Having produced this powerful device, capable of exploiting the useful programming features of the 8080 and circumventing the clumsy ones, Zilog took the surprising step of making the Z80 completely incompatible with the 8080 at the symbolic level. A bizarre



and verbose language was chosen to describe the machine in manufacturer's literature. While this language has undoubted merits in terms of similarity to formal notation used in system software design, there are overpowering objections to it from the point of view of applications programming. What makes a good formal design notation does not make a workable programming language. The needs of the two environments are completely different. Further, this choice of language voids the body of expertise built up so slowly and painfully with the 8080, forcing Z80 users to undergo a completely unnecessary and wasteful relearning task. Finally, as practical objections, the language requires about a third again as many keystrokes per line as conventional languages, and processors capable of assembling this language are not generally available. The consequence of this last fact is that most small system users are employing the Z80 as an 8080, the increment in power going mostly unexploited.

It can be argued that the language specified by the hardware manufacturer forms in some sense a standard. In a field of arbitrary "standards" the manufacturer's arbitrary "standard" might be considered the least arbitrary. The originator of the hardware has the right to describe it in any terms he sees fit. Still, if one chooses to send a shipload of Swahili bibles to Sweden, one ought not to be astonished if the books do not rivet the attention of the Swedes.

In a practical way, the real standard is that which exists, that which is already in common use and with which the largest number of potential users is already familiar and comfortable, i.e., the language of the 8080 family machines. Arguments can be made, many valid, that this language is a hodgepodge of different notations having very little internal consistency. So does the English language, but it is a useful, flexible tool, and attempts to replace it with something closer to the heart's desire of a grammarian would be foolish, as Esperanto proponents have found out.

Obviously the greatest service to the greatest number of users and potential users of the Z80 can be performed by allowing the machine to be programmed in a language compatible with that of the 8080. For this reason the language chosen for use in this book is an extension of the 8080 language, and the assembler in appendix B processes this extension. Using this system, those familiar with the 8080 can continue to program in the language with which they have become comfortable, extending their expertise into the Z80 superset as they

## XII

master each new class of instructions. The choice of language is therefore entirely pragmatic. A cross index of the mnemonics used here with those of Zilog is supplied in appendix D.

The assembler will run on any 8080 family or Z80 machine, being written entirely in the 8080 subset. A Z80 based machine is not required. The program requires somewhat less than 10K RAM, the precise amount depending on the length of the user supplied input/output routines. It will process source text into object code at about 1000 lines per minute on a system running at 2MHz. As some readers may wish to make changes in the assembler, the methods used in it are the simplest possible, at the cost of some space and speed. While no program of this length and complexity can be certified to be free of error, the assembler and debugging monitor have been subjected to extensive use and testing and contain no known errors. If any are discovered by readers, either in the software or the text, the writer would greatly appreciate being informed so they may be corrected in future printings. Suggestions for improvements and additions are also welcome.

A number of individuals and organizations have contributed to this book, either through discussions and suggestions, testing and criticism of software or provision of materials. They are, alphabetically:

Mr. Ralph Hayford  
Mr. Guy Hobart  
Mr. Larry Leske  
Mr. Harvey Nice  
Mr. William Powers  
Mr. Harold Scoblow  
Mr. Albert Shatzel  
Mr. Ted Singer  
Mr. Mel Thomsen  
Victor Comptometer Inc., Components Division  
Mr. Karl Weller  
Zilog, Inc.

*W. Weller  
Chicago  
July 1978*

*Z80 is a registered trademark of Zilog, Inc.*

# 1 / THE NATURE OF THE PROGRAMMING TASK

“The idea of non-human devices of great power and great ability to carry through a policy, and of their dangers, is nothing new. All that is new is that now we possess effective devices of this kind. In the past, similar possibilities were postulated for the techniques of magic, which forms the theme for so many legends and folk tales. In all these stories the point is that the agencies of magic are literal-minded; and that if we ask for a boon from them, we must ask for what we really want and not for what we think we want. The new and real agencies of the learning machine are also literal-minded. If we program a machine for winning a war, we must think well what we mean by winning. We can fail in this only at our immediate, utter, and irretrievable peril. We cannot expect the machine to follow us in those prejudices and emotional compromises by which we enable ourselves to call destruction by the name of victory. If we ask for victory and do not know what we mean by it, we shall find the ghost knocking at our door.”

Norbert Wiener  
*Cybernetics* 1948

To say that programming is a difficult activity is to utter a platitude. The great amount of literature and diverse opinion about methods and languages are testimony enough to the exact and exasperating character of an activity which is almost never performed correctly the first time. It is the purpose of this introductory chapter to attempt to explain to the reader why this is so, and by so explaining to help him avoid some of the pitfalls inherent in the task.

If the essence of the problem has to be condensed into a single sentence, the sentence would read something like this: *programming computers is an alien psychological task*. It is communication of a sort, but of a sort to which our associative mental processes are badly adapted. Learning to do it can be compared in some ways to learning a foreign language. It involves a vocabulary and a set of rules

which must simply be memorized, along with the exceptions to the rules. When this has been accomplished a long and tedious period of practice follows, involving bungling and false starts. This period of practice leads by small increments to fluency in the use of the new language. If a second foreign language is to be learned the task is a bit easier, since some of the knowledge is "portable". Similarities in structure and vocabulary are noted and remembered, these similarities working as "bootstraps" in beginning with the new language.

The similarity ends here, however. Learning a foreign language is simply learning an alternate form of human communication. While the various forms of human communication look very different, they are really much more similar than different. The nuclear ideas of human communication are universal, concerned as they are with things which concern the human beings using the communication system known as language. Fundamental to human communication is the notion of implication, e.g., the remark by a Soviet politician that leaving the West to police itself in certain matters was "leaving the goat to guard the cabbage". Mr. Khrushchev did not have to say that he distrusted the West; the implied parallel with the goat and the cabbage did it for him.

Human communication with a machine is another matter entirely. There are no implications or nuances of expression. The device is, to use Wiener's expression, totally literal-minded, and dealing with a literal-minded agency is a task for which we are not by nature equipped, and our folk lore is rich in stories which illustrate our awareness of it. There are many such stories but perhaps the most vivid of them is called "The Monkey's Paw" by W.W. Jacobs. In this story an elderly English couple comes into possession of a magic piece, brought back from India by a member of the British Army stationed there. The magic piece, a monkey's paw, allows them three wishes, the first of which is for a large sum of money. This wish is granted when their son is mangled to death in machinery and the insurance company pays off his life policy. The second wish is for the return of their son to life. This is granted when they are tortured by the sound of his mangled body hammering at their door in the middle of the night. The third wish is used to return the son to his grave.

The point of this and all such stories is that the magic agent is completely literal-minded, as is an automatic device like a computer. This was the characteristic of computers that bothered Wiener in the



quote at the head of this chapter. Does victory mean that one more of us is left alive than of them? The logical outcome of this definition of victory is a single human left alive. Is this what we really mean by victory? I think not, and become very uneasy at the idea of offensive and defensive weapons systems under the control of programming about which I know nothing.

As Wiener says, the only difference is that we now possess effective devices of this kind. We must learn to deal with them, learning to instruct them in a totally literal way, and in this completely literal character of the instruction lies the psychological alienness of the task. If this is only well understood the reasons for the failures to solve the problem of programming difficulty become immediately obvious. While one computer language form may differ from another in its superficial aspects (compiler, assembler, etc.) *all* of these language forms without exception require that we conceptualize the problem in totally literal, discrete steps. If the problem to be solved can be conceived of in this way then the difficulty vanishes. After this has been done the language in which the problem is programmed is pretty much irrelevant. Different languages offer facilities for convenient expression of problem solutions of different classes. Some of these languages succeed and others fail. The most long lived and successful of the higher level languages, FORTRAN, succeeded not because it relieved the programmer of the job of correctly conceptualizing the problem to be solved, but because it addressed a class of problems (mathematical) which had been reduced to procedural form long before the invention of the first computer. Relatively speaking, they were the easy problems. FORTRAN did not perform well for the problems of business, and other language forms were invented for these applications, eventually converging into COBOL and RPG. COBOL and RPG succeeded because they addressed a set of problems which were already well defined and reduced to procedural form. All of this is by way of saying that a programming language is an effect, not a cause. It exists as a response to the development of the ability to define a class of problems in precise procedural terms. It is the confusion of cause and effect which has led to the appearance and disappearance of so many programming languages over the years. Most of them were cures for diseases which did not exist, or which existed in such minor form that the language which addressed them had negligible application. The central nub of the

problem of programming and the nucleus of the difficulty of the activity is the need to define the problem to be solved in discrete, totally literal steps, and any system of programming which purports to relieve the programmer of this responsibility misrepresents itself. This is most eloquently put by Norbert Wiener:

“No, the future offers very little hope for those who expect that our new mechanical slaves will offer us a world in which we may rest from thinking. Help us they may, but at the cost of supreme demands upon our honesty and our intelligence. The world of the future will be an ever more demanding struggle against the limitations of our intelligence, not a comfortable hammock in which we can lie down to be waited upon by our robot slaves.”

Norbert Wiener  
God & Golem, Inc. 1964

The problem thus reduces to finding some way to match a human mind which operates in a fuzzy, associative way, with a robot which will tolerate no fuzziness and is incapable of association. A fundamental adaptation must be made by one or the other. Since the computer cannot adapt because we do not yet know how to build a machine capable of adaptation, it is the human mind which must stretch to bridge the gap. That this is possible is testified to by the many successful existing applications of computers, but it is not easy, and the road to any successful computer application is littered with potholes into which the unwary will fall. Computer programming, like any other significant skill, is built heavily on experience. Good programmers are made, like good brain surgeons and engineers, by experience, i.e., by making mistakes, finding the reason for the failure, fixing it and *remembering the mistake so it isn't repeated*.

It is tempting, when dealing with a particularly exasperating mistake, to claim machine failure. This may save the ego for the moment, but it is almost never true. In the writer's experience, which includes some of the early vintage vacuum tube machines, only once has an error turned out to be traceable to machine malfunction. While this may be partly due to good fortune, it remains true that errors caused by the programmer far outnumber those which might be caused by the machine.

Having so described the dimensions of the programming problem, there remains the question of getting the necessary “hands on” expe-

rience to acquire the skill. The reader is urged at this point to follow the directions in appendix A for initializing the assembler and debugging monitor. Using these two programs, the various example programs in the text can be tried out as they are studied. Only in this way can a real understanding of their functions be achieved. The operating instructions for the assembly program are in chapter 4, and those for the debugging monitor in chapter 17. Those already familiar with operations on binary quantities and the general organization of a computer can skip directly to chapter 4 and begin. If you are fuzzy on these topics by all means do not skip these chapters. Understanding their contents is vital to successful programming of the Z80 or any other computer.

## 2 / BINARY OPERATIONS

“We have seen that the symbols of logic are subject to the special law  $x^2 = x$ . Now of the symbols of Number there are but two, viz, 0 and 1, which are subject to the same formal law. We know that  $0^2 = 0$  and that  $1^2 = 1$ ; and the equation  $x^2 = x$ , considered as algebraic has no other roots than 0 and 1. Hence, instead of determining the measure of formal agreement of the symbols of Logic with those of Number generally, it is more immediately suggested to us to compare them with symbols of quantity *admitting only of the values of 0 and 1*. Let us conceive, then, of an Algebra in which the symbols  $x, y, z$ , etc. admit indifferently of the values of 0 and 1, and of these values alone. The laws, the axioms, and the processes of such Algebra will be identical in their whole extent with the laws, and axioms and the processes of an Algebra of Logic. Difference of interpretation will alone divide them. Upon this principle the method of the following work is established.”

George Boole

*An Investigation of the Laws of Thought* 1854

The fact that computers are built of components which can exist in only two electronic states makes itself felt in almost all transactions between programmer and computer. It is the purpose of this chapter to give the beginning programmer an understanding of the workings of a two state system of arithmetic and other operations which take place within the computer. The two states may be given any desired set of names, on an off, yin and yang or Romeo and Juliet, but they are conventionally referred to as one and zero. Sometimes the zero state is called reset and the one state set.

A system for the expression of information which uses only two states may seem strange at first, but if the reader reflects on the matter he will realize that he has encountered such systems before. Traffic is controlled by a system of lights that have two states, green and red. The intermediate yellow is not a state at all, merely a warning that a change of state is about to take place. Morse code is a system of communication in which information is transmitted by means of



combinations of two states, dots and dashes. An earthier form of two state communication is implied in the statement: "If the shade is up don't come in. My husband is home". Paul Revere's "one if by land and two if by sea" is another example. We deal with two state codes all the time almost without realizing that we are doing so. What is unfamiliar is the idea of expressing numerical information in a two state code.

This is the natural consequence of being born with ten fingers. There is a certain subjective "rightness" about counting by tens and computing in base ten. This feeling of "rightness" has no objective base, of course, as a look into history quickly shows. In ancient times twenty was commonly used as a number base in parts of the world in which the climate allowed people to go barefoot or wear open sandals. Some Europeans of the middle ages and later used twelve as a number base, the traces of this number base still showing in modern language in such terms as dozen and gross.

This feeling of familiarity is quite superficial, and this can be easily shown with young children who have not had enough time to accumulate the necessary prejudice. As we will see in a few pages, it is possible to count up to 31 using only five digits in base two. The five digits can be represented by the fingers of one hand. The letters A through Z can thus be set into one to one correspondence with base two numbers, A being one, etc. Even very young children, as young as seven, catch on to this system of communication very quickly. All that is required is to know how to spell the necessary words and count. The message is spelled out as a series of base two hand signals, an upraised finger being a one and a lowered finger being a zero. A seven year old can be conversing quite fluently in this system in only 15 minutes or so, though the representation of the letter "D" has caused some problems by its unfortunate coincidence with another type of hand signal.

As with decimal and all other number systems the position of a digit in a binary number has place value, the values of successively higher digit positions being the successive powers of the number base. The decimal number 2937, for example, really means:

$$2 \times 10^3 + 9 \times 10^2 + 3 \times 10^1 + 7 \times 10^0$$

the value of  $10^0$  being one. Similarly the base two or *binary* number 10101 has the meaning:

$$1 \times 2^4 + 0 \times 2^3 + 1 \times 2^2 + 0 \times 2^1 + 1 \times 2^0$$

the value of  $2^0$  again being one. If the reader is unfamiliar with this zero power business it is fairly easy to show from the laws of exponents. The result of the division:

$$\frac{a^m}{a^n} = a^{m-n}$$

If the two exponents are identical, the division becomes:

$$\frac{a^m}{a^m} = a^{m-m} = a^0$$

But since the divisor and dividend are equal the result of the division must be one, since dividing anything by itself results in a quotient of one. Any number to the zero power is therefore one.

The decimal value of the base two number shown above is easy to compute when written in this expanded form. The terms containing zeros are ignored, just as in decimal, and the values of the remaining terms are added, namely the  $2^4$ ,  $2^2$  and  $2^0$  terms. Since the only possible multiplier for these terms is one, not 1 through 9 as in decimal, this amounts to simply adding:

$$2^4 + 2^2 + 2^0 = 16 + 4 + 1 = 21$$

As a convenient reference, the first few powers of two are shown in tabular form below.

$$\begin{aligned} 2^0 &= 1 \\ 2^1 &= 2 \\ 2^2 &= 4 \\ 2^3 &= 8 \\ 2^4 &= 16 \\ 2^5 &= 32 \end{aligned}$$

$$2^6 = 64$$

$$2^7 = 128$$

$$2^8 = 256$$

$$2^9 = 512$$

$$2^{10} = 1024$$

$$2^{11} = 2048$$

$$2^{12} = 4096$$

$$2^{13} = 8192$$

$$2^{14} = 16384$$

$$2^{15} = 32768$$

$$2^{16} = 65536$$

The conversion of a longer binary number is shown in example 2-1.

#### Example 2-1

Using the above table convert the binary number 11001001 to decimal. The number is first written in the expanded form shown above, i.e.:

$$1 \times 2^7 + 1 \times 2^6 + 0 \times 2^5 + 0 \times 2^4 + 1 \times 2^3 + 0 \times 2^2 + 0 \times 2^1 + 1 \times 2^0$$

Dropping the terms which contain zero multipliers and substituting the values of the powers of two from the above table, we have:

$$128 + 64 + 8 + 1 = 201$$

This process of conversion will be tedious at first but with a little practice it will become familiar.

Before leaving example 2-1 take note of the fact that the binary number 11001001 required eight digit positions, while its decimal value, 128, required only three. Binary numbers are long, tedious things and working with them in this form is a quick way to introduce errors. For this reason programmers use shorthand methods for writing binary. These are called the octal (base 8) and hexadeci-

mal (base 16) systems. Computation is almost never done in these systems but it is necessary to know how they are represented.

Conversion between binary and octal is simply a matter of grouping the binary digits (bits) and then reading off the decimal equivalents of each group. The binary number in example 2-1, for example, is converted to octal by splitting it into groups of three bits, beginning at the right, like this:

11 001 001

The decimal equivalent of each group is then read off. The leftmost group is a 3, and the other two groups are both 1's. The result is then written:

$$11001001_2 = 311_8$$

the subscripts indicating the number bases. A longer conversion of this type is shown in example 2-2.

#### Example 2-2

Convert the binary number 0001100110011001 to octal. The number is divided into groups of three bits, beginning at the right, the decimal equivalent of each group written underneath, like this:

|   |     |     |     |     |     |
|---|-----|-----|-----|-----|-----|
| 0 | 001 | 100 | 110 | 011 | 001 |
| 0 | 1   | 4   | 6   | 3   | 1   |

the result being:

$$0001100110011001_2 = 014631_8$$

Note here that since the number of bits in the binary number was sixteen, not a multiple of three, the leftmost group has only one bit.



The conversion from octal to binary is equally simple. Each of the octal digits is simply expanded to three bits, as shown in example 2-3.

### Example 2-3

Convert the octal number 72613 to binary. This is done by spreading out the octal digits and writing the binary equivalent of each underneath.

|     |     |     |     |     |
|-----|-----|-----|-----|-----|
| 7   | 2   | 6   | 1   | 3   |
| 111 | 010 | 110 | 001 | 011 |

or:

$72613_8 = 111010110001011_2$

The note at the end of example 2-2 about the dangling lefthand bit brings up a point about the octal number system which must be considered before using it as a shorthand for binary in programming. Computers in general group bits into units whose size is an integral power of two, i.e., 4, 8, 16, or 32 bits. Since none of these groupings is an even multiple of three there will always be a leftmost bit group containing less than three bits if octal is used. It is this fault of the octal system which leads to consideration of an alternate shorthand for binary, the hexadecimal system.

Conversion of binary to hexadecimal is accomplished by grouping the bits into fours, then reading off the value of each, as with octal. The only problem is that four bits can represent a number whose value is greater than nine. Since we have no single symbols to represent these numbers in the decimal system, we simply invent symbols — the first six letters of the alphabet. The symbols used for hexadecimal conversion are shown in table form below.

| <i>Binary</i> | <i>Hexadecimal</i> |
|---------------|--------------------|
| 0000          | 0                  |
| 0001          | 1                  |
| 0010          | 2                  |
| 0011          | 3                  |

|      |   |
|------|---|
| 0100 | 4 |
| 0101 | 5 |
| 0110 | 6 |
| 0111 | 7 |
| 1000 | 8 |
| 1001 | 9 |
| 1010 | A |
| 1011 | B |
| 1100 | C |
| 1101 | D |
| 1110 | E |
| 1111 | F |

Conversion between binary and hexadecimal can be done by referencing the above table. The binary number of example 2-1, 11001001, is converted by dividing it into two groups of four bits each, as:

1100 1001

The rightmost group is simply a nine, an eight bit plus a one bit. The left group is a twelve, however an eight bit and a four bit. Looking in the table above, we see that the symbol for twelve is C. The result is then:

$$11001001_2 = C9_{16}$$

While the alphabetic symbols may give you trouble for a while the system becomes familiar with a little practice and it offers important advantages over octal when dealing with computers like the Z80, as we will see. A more elaborate conversion is shown in example 2-4.

#### Example 2-4

Convert the binary number 0111111010110100 to hexadecimal. The number is written in four four bit groups, like this:

0111 1110 1011 0100

The leftmost group can be read immediately as a 7. The second group from the left has value 14 for which the symbol in the table is E. The third group has value 11, symbol B. The rightmost group can be read directly as a 4.

The simplest possible operation performed by a computer, at root the only operation of which it is really capable, is the comparison of one bit to another, setting another bit or bits to reflect the result of this comparison. All of the more complex operations are built up out of these simple bit comparisons. One type of bit comparison operation is known as the Inclusive OR. In this operation two bits are compared and the result set to a one if either or both of the bits being compared is a one. Like the operations of arithmetic, the Inclusive OR operation has a sign, V. To indicate the Inclusive OR of two quantities they are written with the "V" between them. All of the possible cases are:

$$1 \text{ V } 1 = 1$$

$$1 \text{ V } 0 = 1$$

$$0 \text{ V } 1 = 1$$

$$0 \text{ V } 0 = 0$$

The result of an Inclusive OR can be zero if and only if *both* of the bits being ORed are zero. In the Z80 and other computers bits are ORed in groups rather than singly. In the Z80 eight bits can be handled in parallel at one time. This operation is shown in example 2-5.

#### Example 2-5

Form the Inclusive OR of the eight bit binary quantities 10110101 and 11000100.

The numbers are written one below the other:

$$\begin{array}{r} 10110101 \\ V \ 11000100 \\ \hline \end{array}$$

The work may go from left to right or right to left but we will go from right to left to preserve consistency with the direction used in addition and subtraction operations to come later. To avoid excess verbiage in describing the bit positions within the eight bit group we will number the bits 0 through 7, 0 meaning the rightmost and 7 the leftmost bit. The operation is begun by ORing the 0 positions. The top bit is a one and the bottom a zero. The result is therefore one, written below, like this:

$$\begin{array}{r} 10110101 \\ V \ 11000100 \\ \hline 1 \end{array}$$

Next the bits in the 1 position (second from right) are ORed. They are both zero so the result is a zero, again written below, as:

$$\begin{array}{r} 10110101 \\ V \ 11000100 \\ \hline 01 \end{array}$$

This right to left sweep is continued until all eight positions have been ORed and the results written below. The final result looks like this:

$$\begin{array}{r} 10110101 \\ V \ 11000100 \\ \hline 11110101 \end{array}$$

The numbering of the bits from right to left in example 2-5 was not randomly chosen. Notice that the number of the bit position cor-



responds to the power of two represented by that position. The operation performed in example 2-5 is also known as the *logical sum*. When the single word OR is used from this point on in this book it is the Inclusive OR function which is meant. This is to distinguish it from another type of OR operation which we will consider in a little while.

Another type of bit comparison operation is known as the AND, also sometimes called the *logical product*. In the AND operation a pair of bits is compared and the result set to one if and only if *both* of the bits being compared are ones. If either or both bits are zero the result is a zero. It is fairly easy to see why this is called the logical product. The result is the same as if the two bits were multiplied. The AND operator is an inverted "V". All possible cases for AND are shown below.

$$1 \wedge 1 = 1$$

$$1 \wedge 0 = 0$$

$$0 \wedge 1 = 0$$

$$0 \wedge 0 = 0$$

As with the OR function, the AND is performed on the Z80 eight bits at a time. The operation is shown in example 2-6.

#### Example 2-6

Form the AND of the two binary quantities in example 2-5.

#### Example 2-5

$$\begin{array}{r} 10110101 \\ \wedge 11000100 \\ \hline 10000100 \end{array}$$

Note that ones appear in the result if and only if both of the bits above were ones. If either is a zero the result is a zero. The AND operation is usually used in an operation

called *masking*. The purpose of masking is to isolate some group of bits within a larger group by setting the irrelevant information to zeros. This is done by forming a mask to be ANDed with the information, this mask containing ones only in the bit positions to be preserved and zeros elsewhere. To isolate the lower four bits of the 10110101 above a mask is constructed with ones in the low bits and zeros in the remaining bits. The operation looks like this:

$$\begin{array}{r} 10110101 \\ \wedge 00001111 \quad \text{the mask} \\ \hline 00000101 \end{array}$$

Do not leave this example without understanding it. This operation will appear many times in this book. The AND operation is also sometimes called the *logical product*.

Yet another type of comparison operation is the Exclusive OR, also called the *logical difference*. In this operation the result is a one if and only if the bits being compared are different. The symbol for Exclusive OR is a crossed V, i.e.,  $\nabla$ . All possible cases of Exclusive OR are shown below.

$$\begin{array}{l} 1 \nabla 1 = 0 \\ 1 \nabla 0 = 1 \\ 0 \nabla 1 = 1 \\ 0 \nabla 0 = 0 \end{array}$$

Again, the result is a one only if the bits being Exclusive ORed are different. Notice here that the Exclusive OR function is the same operation performed on algebraic signs of numbers when multiplying or dividing. If a one represented minus and a zero plus, the result of Exclusive ORing will correctly predict the sign of a product or quotient, the algebraic rule being that the result will be minus only if the signs of the operands are different. The Exclusive OR of two eight bit quantities is shown in example 2-7.

## Example 2-7

Form the Exclusive OR of the two eight bit binary quantities in example 2-5.

$$\begin{array}{r} 10110101 \\ \vee 11000100 \\ \hline 01110001 \end{array}$$

The Exclusive OR operation is an efficient way to test two sets of bits for equality. If the two quantities are identical the result of an Exclusive OR will be all zeros.

A logical operation which is performed on a single bit rather than a pair of bits is known as complementation. Complementation means the simple inversion of the value of the bit. If the bit is one its complement is zero. If the bit is zero its complement is one. Complementation is usually indicated by a bar over the quantity to be complemented, i.e.:

$$\bar{1} = 0 \quad \text{and} \quad \bar{0} = 1$$

As with the other logical operations, complementation is performed on eight bits at a time in the Z80. This operation is shown in example 2-8.

## Example 2-8

Form the complement of the eight bit number 10101010. The complement is formed by simply inverting the value of each of the bits, ones becoming zeros and zeros becoming ones. The complement of 10101010 is therefore 01010101. Note that if this result is itself complemented the

original number comes back, i.e., the complement of 01010101 is 10101010.

The complementation process just described is also known as *one's complementation* to distinguish it from another type of complementation which we take up a bit later in this chapter.

Another operation performed on a group of bits is known as *shifting*. To picture what goes on in a shift imagine a section of conveyor which is divided into eight discrete spaces separated by barriers. A space may contain a box (a one) or no box (a zero). The conveyor is capable of being moved only by integral spaces, i.e., it cannot be moved half a space or one and a half spaces, only one space, two spaces, etc. Now move this imaginary conveyor one space to the left. If the leftmost space contained a box (a one), the box falls off the end, and a new empty space enters from the right. If the conveyor spaces were numbered 0 through 7 as in example 2-5, the content of space zero would move to space one, that of space one would move to space two, etc.

If the conveyor had been shifted to the right just the opposite effect would have taken place. The content of space zero would fall off the end and an empty space (a zero) would have been introduced on the left, with the new content of space seven moving to space six, etc. The left shift is shown in example 2-9.

#### Example 2-9

Perform a left shift of the eight bit number 00111111. The shift can be done on paper by simply writing a zero at the right hand end and omitting the leftmost bit, i.e.:

01111110

Before leaving this example, note one important effect of a left shift. The value of the original number, 00111111<sub>2</sub>



was  $63_{10}$ . The value of the shifted number,  $01111110_2 = 126_{10}$ , double the original value. This illustrates an important effect. *If no significant bits are shifted out by a left shift, the effect of the shift is to multiply the original number by 2.* This property of a left shift will be used many times in the remainder of this book.

A right shift is illustrated in example 2-10.

#### Example 2-10

Perform a right shift of the eight bit number  $00110001$ . This shift can be done easily by adding a zero to the left hand end of the group and omitting the rightmost bit, i.e.,  $00110001$  becomes  $00011000$ , the bit shifted out having been a one. Now look at the values of the original number and the shifted number.  $00110001_2 = 49_{10}$  while  $00011000_2 = 24_{10}$ . The effect, in this case, is to divide the original number by two, ignoring any remainder. This will not be strictly true when we come to deal with negative numbers. The general statement of the effect of a right shift is this. *Shifting right one bit has the effect of dividing by two and rounding in the direction of the next more negative number.* This effect will also be used many times in the remainder of this book.

The operations so far discussed can now be combined to perform some simple arithmetic. The sum of two binary digits, called the addend and the augend, can have only four outcomes. These are:

- |             |                                             |
|-------------|---------------------------------------------|
| $0 + 0 = 0$ | with no carry                               |
| $0 + 1 = 1$ | with no carry                               |
| $1 + 0 = 1$ | with no carry                               |
| $1 + 1 = 0$ | with a carry into the next higher order bit |

Ignoring carries into higher order bits for a moment, these sums are exactly the same result which would be gotten by performing the Exclusive OR of the bits being added. It is for this reason that the Exclusive OR operation is also sometimes known as *half add*. In the single case in which a carry can be generated, both of the bits being added must be ones. This corresponds exactly to the result obtained with an AND operation. Thus the sum column can be generated by an Exclusive OR of the bits, while the carry into the next higher order column can be generated by an AND. In the higher order columns the carry out of lower orders must be accounted for. This changes things somewhat, since now there is the sum of three bits to be considered. The possible cases are these:

| <i>carry in</i> | <i>addend bit</i> | <i>augend bit</i> | <i>sum bit</i> | <i>carry out</i> |
|-----------------|-------------------|-------------------|----------------|------------------|
| 0               | 0                 | 0                 | 0              | 0                |
| 0               | 0                 | 1                 | 1              | 0                |
| 0               | 1                 | 0                 | 1              | 0                |
| 0               | 1                 | 1                 | 0              | 1                |
| 1               | 0                 | 0                 | 1              | 0                |
| 1               | 0                 | 1                 | 0              | 1                |
| 1               | 1                 | 0                 | 0              | 1                |
| 1               | 1                 | 1                 | 1              | 1                |

This is the general case of addition of addend and augend bits including carry in from a lower order. In the case of the rightmost bit, of course, the carry in is always zero for a simple add.

The above rules make it possible to perform eight bit addition using Exclusive ORs to form the sum bits, ANDs to form the carries and shifts to move the carries into the next higher order columns. The carries are then again Exclusive ORed with the half sum produced by the previous Exclusive OR. This is repeated until the carries are all zero. The process looks like this.

```

00001001  the augend, a nine
00000111  the addend, a seven
-----

```

First form the Exclusive OR of addend and augend. This is:

```

00001110

```

Then form the AND:

00000001

Since the result of the AND is not zero, there is a carry. The carry or carries are shifted into the next column by performing a left shift, i.e. the result of the AND becomes:

0000010

The result of the Exclusive OR becomes the new addend and the shifted result of the AND becomes the new augend and the process is repeated.

$$\begin{array}{rcl}
 & 00001110 & \\
 \forall & 00000010 & \text{and} \quad \Delta \\
 \hline
 & 00001100 & \begin{array}{r} 00001110 \\ \hline 00000010 \\ \hline 00000010 \end{array}
 \end{array}$$

Since the result of the AND is not zero, it is shifted left again and the process is repeated, i.e.:

$$\begin{array}{rcl}
 & 00001100 & \\
 \forall & 0000100 & \begin{array}{r} 00001100 \quad \text{Exc. OR result} \\ \hline 00000100 \quad \text{AND result, shifted left.} \\ \hline 00000100 \end{array}
 \end{array}$$

The result of the AND is again nonzero, so we shift it left and try again.

$$\begin{array}{rcl}
 & 00001000 & \\
 \forall & 00001000 & \begin{array}{r} 00001000 \\ \hline 00001000 \\ \hline 00001000 \end{array}
 \end{array}$$

Still no zero in the AND result, so we shift it left and try again:

$$\begin{array}{rcl}
 & 00000000 & \\
 \forall & 00010000 & \begin{array}{r} 00000000 \\ \hline 00010000 \\ \hline 00010000 \end{array}
 \end{array}$$

The AND result is now a zero, so the final result of the process is the outcome of the last Exclusive OR. The value of this number, 00010000, is  $16_{10}$ , the correct sum of the original numbers, 9 and 7.

Subtraction can be done by a process implemented in a similar way. It is performed on a pair of bits called the *minuend*, the number from which something is being subtracted, and the *subtrahend*, the number which is being subtracted from the minuend. The four cases of simple subtraction are:

|         |                          |
|---------|--------------------------|
| $0-0=0$ | no borrow                |
| $1-0=1$ | no borrow                |
| $0-1=1$ | borrow from higher order |
| $1-1=0$ | no borrow                |

The result bit, called the difference, is again a one if the minuend and subtrahend bits are different, just as in addition. This means that the Exclusive OR can again be used to form this partial result. The borrow bit becomes a one only if the minuend bit is zero and the subtrahend bit a one. If the borrow bit is represented by B, the minuend bit by M, and the subtrahend bit by S, the condition for a borrow is:

$$B = \overline{M}AS$$

As subtraction is usually done on paper, the borrow is effected by decreasing the next higher order minuend digit by one, i.e., decrementing it. The same effect can be gotten by incrementing the next higher subtrahend digit before proceeding to that column, and this is indeed what is done. The reader might find it an instructive exercise to go through a binary subtraction on paper using this method.

Negative numbers are represented on the Z80 computer by means of their *two's complements*. The two's complement of a number is formed by first taking the one's complement as shown earlier in this chapter, and then adding one. The process is shown in example 2-11.



## Example 2-11

Find the two's complement of the eight bit number 00001011. First the one's complement is found by inverting every bit, i.e.:

11110100

and then one is added to this one's complement:

$$\begin{array}{r} 11110100 \\ + \quad \quad 1 \\ \hline 11110101 \end{array}$$

Since the decimal value of the original number was  $11_{10}$ , this result ought to represent  $-11_{10}$ . This is easy enough to check. If we simply add the original number to its two's complement we ought to get zero.

$$\begin{array}{r} 00001011 \\ + 11110101 \\ \hline 1 \ 00000000 \end{array}$$

the detached one being the carry out of the high order bit sum. This carry is simply discarded. The result is correct.

Before leaving example 2-11, the reader should note something about the number used and its two's complement. The positive or negative character of a number in the two's complement system can be determined by looking at the leftmost bit of the number. If this bit is a zero the number is positive, if a one, negative, but the difference runs deeper than this. In the positive number, 00001011, the highest significant digit was in the  $2^3$  position, i.e., the eight's bit. All the bit positions above this were filled with nonsignificant zeros. In the negative number, however, a significant bit is a zero, not a one. The highest significant bit in the negative number is the *zero* in

the eight's position. The positions above this are filled with nonsignificant ones. In both cases the leading nonsignificant bits are filled with copies of the sign bit, zero for positive and one for negative. While we will speak in this book of the leftmost bit being a sign indicator, it must be understood that *the sign fills all leading nonsignificant bits*, whether positive or negative.

In dealing with any computer, but particularly with machines like the Z80 which handle data eight bits at a time, the generation of a result too large to be held in the allotted space must be considered. This condition is known as *overflow*. It is distinct from and does not have the same meaning as a carry out of the high bit. Overflow occurs when two signed numbers are added or subtracted to give a result which cannot be held in the allotted space. It is most important to understand this, so do not skip over the remainder of this chapter. You do so at your peril.

The largest number which can be held in  $N$  bits is  $2^N - 1$ . For eight bits this means 255, or  $2^8 - 1$ . If the eight bit group has an algebraic sign one bit must be allotted to it, leaving only seven to contain the magnitude of the number. The largest magnitude expressible in seven bits is  $2^7 - 1$  or 127, or in binary:

01111111

If a one is added to this the result is:

$$\begin{array}{r} 01111111 \\ + \quad \quad 1 \\ \hline 10000000 \end{array}$$

which, if considered only as a magnitude, is correct, 128. But since the leftmost bit contained an algebraic sign, this is not correct as a signed result. We have added two positive numbers and gotten a negative sum. This is the overflow condition. Overflow can only occur when numbers of like sign are added or when numbers of unlike sign are subtracted.

Now let us consider the negative case, if we negate (i.e., two's complement) the +127 above we get:

10000001

this being the correct two's complement representation of -127. We will subtract one from this by adding a minus one. The two's complement representation of minus one is 11111111. The reader should verify this by going through the two's complement process with pencil and paper. The addition of -127 and -1 is represented by:

$$\begin{array}{r} 10000001 \\ + 11111111 \\ \hline 10000000 \end{array}$$

but this result is valid. We have added two negative numbers and gotten a negative result. This illustrates something important about the two's complement system. There is one more negative number than positive. The maximum negative number in eight bits is 10000000. If a minus one is added to this an overflow will result, i.e.:

$$\begin{array}{r} 10000000 \\ + 11111111 \\ \hline 01111111 \end{array}$$

Here two negative numbers have been added to produce a positive result, and overflow. The Z80 computer, unlike its predecessor the 8080, has hardware means for detecting the overflow condition. The exact means for using this hardware feature will be explained in a later chapter. The means by which it is detected are of some interest, however.

Overflow is detected by monitoring the carry into the highest bit, i.e., the sign bit, and out of the highest bit. If these two carries are different, then an overflow has occurred. The Z80 has an overflow flag or overflow bit which is set to one when this happens. After an arithmetic operation the condition of the overflow bit is:

$$\text{Carry into sign} \vee \text{Carry out of sign}$$

Recall that the result of an Exclusive OR is a one if and only if the bits are different, and the meaning of this will be clear. The two conditions:

- 1) An add or subtract has produced a result too large to be held in the signed space allotted; and,
- 2) The carry into the sign and the carry out of the sign which resulted from the add or subtract were different,

are equivalent.

The subject of the maximum negative number touched on above requires a little expansion before we move on. This number cannot be negated (two's complemented) since it has no positive counterpart. It is instructive to see what happens if we try to negate it.

|   |          |                                            |
|---|----------|--------------------------------------------|
|   | 10000000 | -128 <sub>10</sub>                         |
|   | 01111111 | the one's complement of -128 <sub>10</sub> |
| + | 1        | plus one for two's complement              |
|   | 10000000 | the same number back!                      |

The result is the maximum negative number, but there is something else. During the addition of the one to the one's complement a one was carried into the sign but a zero was carried out. This is the overflow condition we have just discussed. The Z80 hardware will detect this and set the overflow bit.



### 3 / THE LOGICAL ORGANIZATION OF THE Z80 COMPUTER

“Once the characteristic numbers for most concepts have been set up, the human race will have a new kind of instrument which will increase the power of the mind much more than optical lenses strengthen the eyes and which will be as far superior to microscopes or telescopes as reason is superior to sight.”

Gottfried Wilhelm Leibnitz  
1679

The purpose of this chapter is to present a conceptual picture of a computer which will allow a programmer to work with it in an efficient way. That the organizational ideas presented here do not correspond to the hardware organization precisely will be obvious. The concepts to be understood in this chapter must be viewed in the same way as that in which a navigator, for purely computational purposes, views the stars as being projected onto a sphere of infinite radius. The model does not correspond to the physical truth but provides great operational convenience. The computer model presented here should be viewed in the same way — a useful artifice.

It is productive for a programmer to view the computer as being composed of three distinct logical components; a memory, an arithmetic/logic unit, and a controller or central processing unit (CPU). Memory consists of a device into which information may be placed and from which it can be repeatedly retrieved without destruction, in much the same way that words may be written on paper and read back repeatedly without vanishing from the paper. Writing information into computer memory erases or clears the previous contents. The act of writing information into memory is called *storing*. The meaning of this word is quite specific and unique.

Memory is organized into groups of digits of equal length, each of these groups being known as a *word*. The number of digits or bits

which constitutes a word is known as the *word length*. For the Z80 the word length is eight bits. By coincidence, eight bits is also called a byte, but we will adhere to the more general nomenclature in this book. Each word of memory has associated with it a unique number which identifies it and it alone. This number is known as the *address* of the word. Addresses usually begin with zero and run to the highest number memory word available, without gaps, but this need not be so. The address of a word functions like the number of a post office box. The post office box number has nothing to do with the contents of the box and the address of a memory word has nothing to do with the contents of that word. It is quite important to understand the difference between the address of a memory word and its contents.

The word size is determined by the hardware designer. The programmer can exercise no control over it. This does *not* mean that the Z80 is limited to processing information of length eight bits or less. The operations of the computer allow words to be “chained” end to end, so that arithmetic may be performed on data of any required length.

The arithmetic/logical unit resembles a pocket calculator in its function, except, of course, that its calculation is done in binary. It contains one or more *registers*, temporary storage devices capable of holding one or more computer words. In the Z80 the main operational register is the A register. It is in the A register or accumulator, that most of the arithmetic and logical work of the computer is done, though some arithmetic can be performed in other Z80 registers. Accompanying the arithmetic/logical registers is a group of bits known as the *flag word*. The individual bits of the flag word are set by the Z80 to reflect the outcome of operations performed in the arithmetic/logical registers, e.g., was the result a zero or did an addition or subtraction cause an arithmetic overflow. These flag bits can be individually tested, but more of this a little later.

The computer components discussed so far have no unique properties. The function of memory is the same as that of paper and pencil, while that of the arithmetic/logical unit (ALU) is the same as that of a pocket calculator. What is required to make the paper and calculator work is an intervening “intelligence” which can execute the steps necessary to get a useful result. This set of steps is known as the *program*. The implementation of the individual steps is the function of the controller or CPU. It is the CPU which exercises the su-

pervisory function, driving the other components in such a way as to perform the required task.

The program steps, called *instructions*, reside in memory along with the numbers upon which these instructions are to operate. The CPU fetches the instructions from memory one at a time and supervises their execution. To perform this task it contains two principal registers, the *program counter* or P register and the *instruction* or I register. The program counter contains the memory address of the instruction which follows the one currently being executed. How this comes to be is not important just now, but the fact should be noted and memorized. *The program counter always contains the address of the instruction following the one currently being executed.* In this function the program counter can be said to "point to" the next instruction. This notion of a *pointer* is fundamental to the use of the Z80 or any other computer. Using the program counter as a pointer to the next instruction to be fetched and executed, the basic instruction execution cycle of the computer consists of the following steps:

- 1) The instruction pointed to by the program counter is fetched from memory and placed into the I or instruction register.
- 2) The program counter is incremented by the amount necessary to point to the instruction which follows the one just fetched.
- 3) The instruction in the I register is then interpreted as a command to do something, e.g., move data between memory and ALU registers, test the status of some device external to the computer, perform an addition or subtraction, etc.
- 4) The action implied by the interpretation in step 3 is then taken, i.e., the data are moved, the device status tested, the addition or subtraction done, etc.
- 5) The cycle begins again with step 1 above.

In step 2 above the incrementation of the program counter was deliberately left nonspecific. In the Z80, instructions may be one, two, three or four words long. The program counter is automatically incremented by the amount appropriate to the instruction.

Having discussed these general considerations, we now turn to the specifics of the Z80 computer. Like most microcomputers, the Z80 is organized around a set of registers which are usually thought of as pairs of eight bit registers or single sixteen bit registers. These registers are:

AF — This pair consists of the accumulator previously discussed and a flag word whose bits are set to reflect the outcome of the last operation performed. Only 6 of the 8 bits of the F register are used. The bits are:

S — a one if the last result was negative;

Z — a one if the last result was zero;

A — a one if a carry/borrow occurred between bits 3 and 4 of the A register during the last operation;

V — a one if the last operation resulted in arithmetic overflow;

N — a special flag used during decimal arithmetic operations;

C — a one if a carry or borrow was propagated out of the high bit of A during the last operation.

BC — The B and C registers may be individually manipulated or used together as a pointer to transfer data between the A register and memory.

DE — Same as BC above.

HL — The H and L registers perform the same functions as BC and DE above. In addition, data may be moved to or from memory and the A, B, C, D or E registers using HL as a sixteen bit pointer.

AF' } These four register pairs form a second set which perform  
BC' } the same functions as their nonprimed counterparts above.  
DE' } Only one set of registers may be worked on at a time, but  
HL' } switching from one set to the other is a very simple matter.  
This extra register sets represents a major extension of capability over the Z80's predecessor.

SP — An abbreviation for Stack Pointer. The 16 bit SP register allows a special type of data movement between other registers and memory and between the program counter and memory.

X — The X register is used as a single 16 bit register to transmit information between memory and other registers. The pointer contained in X may be modified by the addition of a signed displacement. This makes X a powerful instrument for addressing arrays of data in memory.

Y — Same function as described for X above. The X and Y registers represent a significant increment of power over the 8080.

The main arithmetic register of the Z80 is A, in which arithmetic/logical operations may be performed on data eight bits at a time. Some sixteen bit arithmetic can also be done using the BC, DE and HL pairs. The ways in which these registers and register pairs are used to perform common programming tasks is the subject of the remainder of this book.



## 4 / MACHINE AND ASSEMBLY LANGUAGE PROGRAMMING

“When I use a word”, Humpty Dumpty said in a rather scornful tone,  
“it means just what I choose it to mean — neither more nor less.”

Lewis Carroll  
Through the Looking Glass

In chapter 3 the concept of an instruction was discussed. The instructions of a program, being held in the same memory as the data upon which they operate, are simply binary numbers. It follows then that it is possible to write a program to perform some useful function entirely in binary, and this is indeed the case. Consider the binary program below:

| <i>hexadecimal<br/>memory address</i> | <i>binary contents</i> |
|---------------------------------------|------------------------|
| 100                                   | 00101010               |
| 101                                   | 00010110               |
| 102                                   | 00000001               |
| 103                                   | 00010001               |
| 104                                   | 00000001               |
| 105                                   | 00000000               |
| 106                                   | 10110111               |
| 107                                   | 11101101               |
| 108                                   | 01010010               |
| 109                                   | 00111000               |
| 10A                                   | 00000100               |
| 10B                                   | 00010011               |
| 10C                                   | 00010011               |
| 10D                                   | 00011000               |
| 10E                                   | 11111000               |

|     |          |
|-----|----------|
| 10F | 11001011 |
| 110 | 00111010 |
| 111 | 11001011 |
| 112 | 00011011 |
| 113 | 11000011 |
| 114 | 00010011 |
| 115 | 00000001 |
| 116 | -----    |
| 117 | -----    |

If this eighteen word program is manually entered into memory, the program counter set to 100 hex and the machine started, the program will take the square root of the sixteen bit number left in memory locations 116 and 117. The problem with this kind of a program is that it has no meaning to a human programmer. The instruction at locations 100, 101 and 102, for example, load the Z80's HL register pair with the contents of memory locations 116 and 117. There is nothing about 00101010 00010110 00000001 which suggests this function. The instruction at 10B adds one to the contents of the DE register pair, but there is certainly nothing about 00010011 which suggests this function. There must be a better way.

We could write a small program which would take typed hexadecimal digits from a keyboard and load them into memory. This would certainly be shorter than keying in directly in binary, but hardly more meaningful. Nothing about 2A1601 suggests loading the HL register pair. This is sometimes done during the program debugging process when a single instruction has to be "patched" into a program. It is not an efficient way to write a program of any size.

This difficulty was understood very early in the craft of programming, of course, and means were invented to solve the problem. Programs were created which were capable of translating specific symbols into binary patterns much as was suggested in the paragraph above. These simple programs rapidly evolved into sophisticated programming tools capable of translating a program written partly or totally in symbols into a machine language program. A program capable of doing this is known as an *assembler* or *assembly program*.

In using an assembly program, the programmer prepares his program using symbols to represent the machine instructions. These

symbols, called mnemonics (pronounced nee-mon-ics, initial m silent), from a Greek word meaning memory, are designed so as to suggest their functions. The A register is loaded by means of LDA and stored by means of STA, for example. The direct load of the HL register pair discussed above is represented by LHLD. The symbolic program is then transcribed onto some machine readable medium. This symbolic program, known as the *source program*, is then processed by the assembler. This processing, called *assembly*, has two results, an *assembly listing* and an *object program*. The object program will be taken up a bit later. An assembly listing of the square root program is shown in example 4-1.

### Example 4-1

```

PAGE 01

001          * SQUARE ROOT PROGRAM
002          ORG  :100      START HERE, BRUNHILDE
003 0100 2A1601      LHL N      GET NUMBER
004 0103 110100      LXI D,1    1ST ODD INTEGER TO D-E
005 0106 B7          RSC        CLEAR CARRY
006 0107 ED52      $ LOOP      DSC D      SUBTRACT ODD INTEGER
007 0109 3804      $          BC END      CROSS ZERO?
008 010B 13          INX D      NO, FORM NEXT ODD
009 010C 13          INX D      INTEGER
010 010D 18F8      $          BRA LOOP    AND DO IT AGAIN
011 010F CB3A      $ END      LRD        DOUBLE SHIFT THE
012 0111 CB1B      $          RER        RESULT. ROOT IN E REG.
013 0113 C31301      HERE      JMP HERE
014 0116 FFFF      N          DBL 65535
015 0118          END

*****
* S Y M B O L   T A B L E *
*****

END      010F      HERE      0113      LOOP      0107      N          0116

```

The assembly listing in example 4-1 looks complicated but we will take it in easy steps. Notice first that the material in the listing is divided into distinct columns. The three digit number in the left hand column is simply the line number. It has nothing to do with the assembly process. it is there purely for editing purposes. Begin by looking at the column which has ORG in line 2. This is the column which contains the mnemonics for the machine instructions. It is known as the *instruction field*. Just beneath the ORG is LHLN. This is the mnemonic for the machine instruction which loads the HL reg-

ister pair discussed before. Loading, of course, implies loading from somewhere and this somewhere is given in the next column to the right. The N in this column is known as the *operand* of the LHLD and the column itself is called the *operand field*. The assembler must have some means of knowing what the programmer means by the N, since N is not a numerical address. On line 14 the column to the left of the instruction field contains the N we are looking for. When a symbol like N is used in this field, called the *label field*, it is said to be *defined*. The symbols LOOP, END and HERE are similarly defined by their use in the label field.

Not all mnemonics require that their operands be defined in this way. The LXI on line 4 has D,1 as its operand. The LXI is another type of load, but one in which the data to be loaded are contained in the instruction itself. The D means the DE register pair. This instruction causes the DE register pair to be loaded with the 16 bit number 1, i.e. a 1 with fifteen leading zeros.

The precise meanings of all mnemonics in example 4-1 are not important now, so don't let the apparent complexity of the listing intimidate you. Some of the things in the instruction field are not instruction mnemonics at all. ORG and END are directives which tell the assembly program how to assemble the source program. The meanings of ORG and END are fairly obvious. ORG (origin) tells the assembly program where in memory the assembled instructions are to start. END is simply a flag to the assembler that there are no more lines to be processed.

So far we have discussed the label, instruction and operand fields. There is another field whose use is optional. This is the *comments field* to the far right of the listing. As its appearance might suggest, it is simply a way for the programmer to make notes to himself about what is going on in the program at this point.

Besides processing symbols into a machine language program, the assembler is capable of detecting certain types of clerical errors in a program. The assembler has no way of knowing whether the logic of a program is correct, but it detects and warns the programmer of certain conditions. There were no detectable errors in example 4-1, but the assembler did flag certain lines with a dollar sign (\$). This dollar sign is a warning to the programmer that the instruction he has written will run only on the Z80, not on the 8080. An assembly with some actual errors is shown in example 4-2.

## Example 4-2

```

PAGE 01

001          * D I S A S T E R
002          ORG :2000
003 2000 3A0620 @ LDA Q
004          S
005          U
006          0
007 2003 000000 STA M
008 2006 00      0 ***
009 2007 00000000 FUZZ
010          0
011 200B 1800 $   BRA :3000
012          A
013 200D 21C2C1 LXI H,'AB'
014 2010 210000 LXI H,'ABC'
015          L
016 2013 00     MOV T,M
017          R
018 2014 DD7E00 $   LAX 300
019          D
020 2017 050AC7 DATA 5,'A','G'
021 201A DATA 078
022          U
023 201A FE00 CPI ''
024          L
025 201C          END

*****
* S Y M B O L   T A B L E *
*****

Q      2006

```

This listing shows some of the errors which the assembler detects, as well as other assembler features. The first line, which contains an asterisk (\*) as the first character of the label field is all commentary. Any line which begins with an asterisk is printed in the assembly listing as is but otherwise ignored by the assembler. The next line, **ORG :2000** tells the assembler that the program is to be located at hexadecimal memory address 2000. The colon which precedes the 2000 indicates that the number is hexadecimal. A similar operand appears in line 7. On line 3 while the **LDA Q** part of the line is legal, the label field contains only an "at" sign (@). Since a label must begin with an alphabetic character this label is not acceptable to the assembler. This is noted on the next line by the "S" (for symbol) flag. Line 4 is correct in form, but the operand **M** has not been defined by use in a label field anywhere in the program. The assembler thus cannot assign an address to the **STA** instruction. **M** is said to be undefined in



this case, this being flagged by the “U” on the next line. Line 5 contains no errors. The triple asterisk assembles as a word of zeros, and would be entirely equivalent to the statement:

**Q      DATA      0**

The use of a triple asterisk is a means of visually flagging cells which are modified by the operating program. Line 6 contains the instruction **FUZZ**. The problem here is that there is no such instruction on the 8080 or Z80. This is flagged on the next line with “O”, meaning operation error. Line 7 contains a legal Z80 (but not 8080) instruction. The problem is that this instruction has a limited area of memory in which it can work and the specified address (:3000) is outside this area. The “A” flag on the next line indicates this addressing error. Lines 8 and 9 contain the same legal instruction. Line 8 is correct, but line 9 specifies ‘ABC’ as part of its operand. This part of the operand is limited to 16 bits, the operand is too long to fit into the allotted space. This is indicated by the “L” (for length) flag on the next line. Line 10 contains a legal 8080 and Z80 instruction, **MOV**. **MOV** means move data, either between registers or between memory and registers. The problem is that there is no such thing as a T register. This results in the “R” flag, meaning register specification error. Line 11 contains a Z80 instruction which refers to the X register. This instruction requires an operand, called a displacement in this case, in the range +127 to -128. Since the 300 is out of this range the instruction is flagged with a “D” for displacement error.

Line 12 contains no errors, but shows the assembler’s ability to generate various types of data. The **DATA** in the instruction column is not an instruction at all, but something called a pseudo operation or more commonly, a *pseudo-op*. **DATA** allows the generation of program constants in a number of forms. The number of operands of a **DATA** pseudo-op is variable, and operands are separated by commas. The first operand, 5, is a simple decimal constant. The second begins with a colon, like the :2000 and :3000 in the lines above. The colon specifies that the number which follows is hexadecimal. The third is a single character set off by apostrophes, the letter G. This generates the ASCII code for the letter G, which is 11000111, or C7<sub>16</sub>. Look at the column of the listing immediately to the left of the label field and the hexadecimal representations of all of these con-

stants can be seen. This field shows the results of assembly of the source statement on that line, but we will return to this in a moment. Line 12 is another **DATA** statement. The “at” sign (@) before the operand specifies that the operand itself is supposed to be octal or base eight. This operand contains the digit 8, however, and there is no digit 8 in the octal system, so the line is flagged with a “U” for unknown field type. Line 14 has the same problem as line 9, but this time the operand is too short. The apostrophes are supposed to enclose a character but they don’t, so this too is flagged as a length (L) error.

Some further features of an assembly listing and capabilities of the assembly program are shown in example 4-3.

### Example 4-3

```

PAGE 01

001                                GRAND EQU 1000
002                                ORG  GRAND
003 03E8 0606                      SUM MVI B,END-BEGIN+1 COUNT TO B
004 03EA AF                        ZAR                      CLEAR A
005 03EB 21F503                     LXI H,BEGIN ADDRESS TO H&L
006 03EE 86                          LOOP ADD M ADD ARRAY MEMBER
007 03EF 23                          INX  H INCREMENT ADDRESS POINTER
008 03F0 10FC $                      DBNZ LOOP AGAIN IF COUNT NOT ZERO
009 03F2 32FB03                     STA  ANSWER
010 03F5 01030507                   BEGIN DATA 1,3,5,7,9
011 03FA 0B                          END  DATA 11
012 03FB 00                          ANSWER ***
013 03FC                             END

*****
* S Y M B O L   T A B L E *
*****

ANSWER 03FB  BEGIN 03F5  END 03FA  GRAND 03E8
LOOP 03EE  SUM 03E8

```

We will skip over the first two lines of this listing for the moment, while we discuss details about the rest of it. Look at the column immediately to the right of the line number column. The hexadecimal number in this column is the memory address to which the first word of the instruction on this line has been assigned by the assembler. If a label has been used on the line, this value is assigned to that label. Look into the symbol table at the bottom of the listing and find the label **SUM**. Immediately to its right is the value of 03E8. In line 6 the

symbol **LOOP** has been assigned the value 03EE. Likewise, the symbols **BEGIN** and **END** have been assigned values 03F5 and 03FA.

Since these symbols have numerical values in the assembly, these values can be manipulated and the program in example 4-3 uses this manipulation. This program adds the values of the six numbers starting at **BEGIN** and going to **END**. While we have not yet covered enough material here to show exactly how it works, it should be fairly obvious that a count of six must be generated somewhere. The instruction in line 3 loads the Z80's B register with the value of the operand field after the comma, i.e. the value of:

$$\text{END-BEGIN} + 1$$

The value of this operand can be determined easily by substituting the values from the symbol table for the symbols. The expression above then becomes:

$$03FA-03F5 + 1$$

If you work out the value of this by performing the hexadecimal subtraction and adding the one, it comes out to six, the number of numbers to be added. The usefulness of this might not be immediately obvious, but think what would happen if the length of the table to be added was to change. Not only would the number have to be inserted in the table, but the count would have to be changed. Generating the count symbolically as in example 4-3 allows the assembler to take care of the clerical detail of adjusting the count.

A very important feature of example 4-3 is on the first line, the one with **EQU** in the instruction field. Up until now the only way to give a symbol a value has been to use it in the label field of an instruction. This is different, however. The **EQU** (equals) generates no words in the program. It is simply a directive to the assembler to consider the value of **GRAND** to be  $1000_{10}$  ( $03E8_{16}$ ). On the very next line the symbol **GRAND** is used in the operand field of an **ORG**, the directive which tells the assembler where this program is to go in memory. The result is that the first word of the program is assembled at  $03E8_{16}$ .

The rules of the assembly program are fairly simple and need not be all memorized at once. They will be mentioned as required in the

text. One thing that should be cleared up before going further is the rule for forming labels. A label may be a string of from one to six characters of which the first *must* be alphabetic. The remainder may be any characters whatever except plus (+), minus (-), or comma. Deviations from these rules will draw a nasty message from the assembler.

There is one difference between our assembler and some other current ones that needs to be emphasized. There are *no* reserved names in the assembler. By this we mean that a name is not excluded from being used as a label because it accidentally may mean something else in another context. In many assemblers the names of machine registers may not be used as labels, i.e., they are reserved for assembler use. No name which satisfies the rules set out in the paragraph above is excluded. If you want to name something A or BC or HL go ahead and do it.

Finally, don't be intimidated by the seeming complexity of all this. With a little practice the uses of these features of the assembly program will become clear. The key to learning assembly level programming is practice and lots of it. In a very real way the sum of your knowledge will be the sum of all your experience, both successful and unsuccessful. Making mistakes is important, since more learning takes place in tracking down mistakes than in simply running successful programs.

## 5 / USING THE ASSEMBLY PROGRAM

“Though this be madness, yet there is method in it.”

William Shakespeare  
*Hamlet* Act II

Now that we know something about the assembly language itself and how it looks, we can turn to the practical problem of how to use it.

The assembler listed in appendix B of this book will run on *any* 8080 or Z80 computer system with enough memory to hold it. The only difference between systems that will concern us here is in the details of the input and output processes. If you have not already done so, this is the time to go to the appendix and make such adjustments to the input and output routines as are necessary to make the assembler run on your configuration of equipment.

Source programs are prepared for assembly using a program known as a *text editor*. The text editor is, in this case, built into the assembler and functions as a part of it. It is the simplest possible program, with the minimum number of commands, that will do what is required of it. This has been done deliberately to minimize the overhead of mental clutter which must be forced on the beginning programmer.

The text editor uses memory space beyond the end of the assembler to hold the text of the source program being developed. The basic unit of text is a line. This simple text editor always handles information a line at a time. The text lines can thus be viewed somewhat like old fashioned printers' type slugs. Each slug represented an entire line, the top slug being line one, the next line two and so forth. If a slug were added after line five, the former line six became line seven, the former line seven became line eight, etc. The text editor works in an exactly analogous way. The area of memory in which the text is held is known as the *source buffer*.



The text editor is started by setting the program counter of your computer to the beginning location of the assembler and starting the computer. The terminal device carriage will return and the greater than symbol (>) will be printed. The greater than symbol signifies the editor's readiness to accept a command. Let's suppose we wanted to enter the text MARY HAD A LITTLE LAMB into the source buffer. Entry of text into the buffer is done with the A command. Every text editor command is terminated by a period. Typing A. after the "greater than" symbol allows the text editor to accept material to be entered into the buffer. Each line is then typed ending with a carriage return. When you have finished typing the lines type a period as the first character of a new line and control will return to the text editor, ready to accept another command. The entry of MARY HAD A LITTLE LAMB looks like this.

```
>A.
```

```
MARY
```

```
HAD
```

```
A
```

```
LITTLE
```

```
LAMB
```

```
.
```

```
>
```

If an error is made in typing under control of the A command, it can be corrected by using the back arrow (←). Typing back arrow has the effect of erasing the character typed just before the back arrow. It can be used to backspace over any number of characters this way, to reach a mistake made early in the line. The correct text from the point of error may then be typed. If the back arrow is typed as the last character of a line, i.e., just before the carriage return, the entire line is ignored.

We have entered each word on a different line for a reason which will become clear in a moment. There are five lines in the source buffer. The text editor can be made to display any of these lines by

means of the L (list) command. The L is followed by the number of the line to be listed. Typing L2. results in:

```
>L2 .  
0002 HAD
```

```
>
```

The line is listed and another “greater than” character typed, meaning that the editor is ready for another command. If more than one line is to be listed, the beginning and ending line numbers are separated by a space. To list lines 2 through 4:

```
>L2 4 .  
0002 HAD  
0003 A  
0004 LITLE
```

```
>
```

To list the entire buffer contents we could type:

```
>L1 5 .  
0001 MARY  
0002 HAD  
0003 A  
0004 LITLE  
0005 LAMB
```

```
>
```

This is easy enough, since the text in the buffer is only 5 lines long. In editing long texts, however, it is very easy to lose track of the number of lines in the buffer. Keeping track of this number is a nuisance so the text editor does it for you. The use of the letter F as a line number means the final line of the buffer, whatever it is. The entire buffer contents can be typed out by:

```
>L1 F.  
0001 MARY  
0002 HAD  
0003 A  
0004 LITLE  
0005 LAMB  
  
>
```

or the last line only by:

```
>LF.  
0005 LAMB  
  
>
```

We have a problem here, however. Most readers will have noticed by now that there is a misspelled word on line 4. Line 4 can be gotten rid of by the D (delete) command, i.e.:

```
>D4.  
  
>
```

After deleting line four the buffer lists this way:

```
>L1 F.  
0001 MARY  
0002 HAD  
0003 A  
0004 LAMB  
  
>
```

Note that with line 4 gone, the former line 5, LAMB, has become line 4. If we had deleted line 1 instead, all of the succeeding lines would have been automatically renumbered. Now to get line 4 back in, this time properly spelled. To do this we again use the A command, but this time with a line number. The effect of using A with a

line number is to add the typed in text *after* the line named in the A command. Thus the new line is to be inserted *after* line 3 of the existing text, i.e.:

```
>A3.  
  
LITTLE  
.  
>
```

and the buffer now looks like this:

```
>L1 F.  
0001 MARY  
0002 HAD  
0003 A  
0004 LITTLE  
0005 LAMB  
  
>
```

Now suppose that something had to be added to the end of this text. The A command can be used with F for a line number just as the L command was. We can add lines by:

```
>AF.  
  
AND  
BOY  
WAS  
THE  
DOCTOR  
UPSET  
.  
>
```

The buffer now looks like this:

```
>L1 F.  
0001 MARY  
0002 HAD  
0003 A  
0004 LITTLE  
0005 LAMB  
0006 AND  
0007 BOY  
0008 WAS  
0009 THE  
0010 DOCTOR  
0011 UPSET
```

```
>
```

The D command can be used to delete a series of lines. For example:

```
>DF.  
>D4 9.  
>
```

This results in the buffer contents:

```
>L1 F.  
0001 MARY  
0002 HAD  
0003 A  
0004 DOCTOR
```

```
>
```

The buffer contents can be punched onto paper tape for reuse later by means of the P command. P works just like L except that the text is sent to the system paper tape device instead of the terminal device. Also there are no line numbers on the paper tape. The contents of the entire buffer can be punched by:

```
>P1 F.
```



After the period is typed the computer will pause until the space bar is struck. The buffer will be punched out and the computer will pause again waiting for the space bar. Striking the space bar this second time causes a "greater than" character to be printed and the editor is ready to accept another command.

The tape so produced can be reloaded into the buffer by the R (read) command. This reads lines from the system paper tape read device and adds them to the end of whatever may already be in the buffer. Suppose that the buffer contained:

```
>L1 F.
0001 SUZY
0002 HAD
0003 A
0004 LAWYER

>
```

and the tape just punched were read by the R command. The result is:

```
>L1 F.
0001 SUZY
0002 HAD
0003 A
0004 LAWYER
0005 MARY
0006 HAD
0007 A
0008 DOCTOR
```

For grammatical correctness we add a line after line 4, i.e.:

```
>A4.
AND
.
>
```

and the buffer reads:

```
>L1 F.  
0001 SUZY  
0002 HAD  
0003 A  
0004 LAWYER  
0005 AND  
0006 MARY  
0007 HAD  
0008 A  
0009 DOCTOR
```

>

If the tape to be read represents an entirely separate piece of text, i.e., a new program, the buffer must first be cleared. This is accomplished by the / command, i.e.:

```
>/.
```

Using the text editor to type in programs differs little from what has already been illustrated in this chapter. Some special abilities of the text editor come into play here however. Turn back to example 4-1 for a moment and notice that the columns are aligned. This alignment is automatic. The programmer does not need to count out a number of spaces before beginning to type the instructions. If the line begins with a blank, a blank label field is implied and the text editor will supply the necessary spacing. Likewise, the text editor can recognize which instructions require operands and which do not. In listing instructions which require no operand, the operand field is left blank and any nonblank text following the instruction is moved to the comment field. This is better understood in the form of an illustration rather than further discussion. The program in example 4-1 is typed in like this:

>/.  
>A.

```
* SQUARE ROOT PROGRAM
ORG :100 START HERE, BRUNHILDE
LHLD N GET NUMBER
LXI D,1 1ST ODD INTEGER TO D-E
RSC CLEAR CARRY
LOOP DSC D SUBTRACT ODD INTEGER
BC END CROSS ZERO?
INX D NO, FORM NEXT ODD
INX D INTEGER
BRA LOOP AND DO IT AGAIN
END LRD DOUBLE SHIFT THE
RER RESULT. ROOT IN E REG.
HERE JMP HERE
N DBL 65535
END
```

but when it is listed, it looks like this:

```
>L1 F.
0001 * SQUARE ROOT PROGRAM
0002      ORG      :100      START HERE, BRUNHILDE
0003      LHLD    N          GET NUMBER
0004      LXI     D,1        1ST ODD INTEGER TO D-E
0005      RSC                     CLEAR CARRY
0006 LOOP   DSC     D        SUBTRACT ODD INTEGER
0007      BC      END        CROSS ZERO?
0008      INX     D          NO, FORM NEXT ODD
0009      INX     D          INTEGER
0010      BRA     LOOP       AND DO IT AGAIN
0011 END    LRD                     DOUBLE SHIFT THE
0012      RER                     RESULT. ROOT IN E REG.
0013 HERE   JMP      HERE
0014 N      DBL     65535
0015      END
```

The text editor has automatically reformatted each instruction for easier reading. Note in lines 2, 3 and 4 that the operand field has been filled in. In line 5, however, the operand has been omitted. **RSC** is a synonym for another instruction which has the effect of resetting (turning off) the carry bit. The point here is that it requires no operand. Anything which follows it must be commentary and the text editor recognizes this.

The programmer is thus freed from the nuisance of counting spaces to align the various fields of a source statement. This is an important feature. Programming is a demanding task which tolerates no error, and automatic elimination of a possible source of error makes the task that much easier.

We have already noted two ways in which comments can be written into a program, beginning the line with an asterisk and using the short comment field of the source line. There is a third way which has a good deal to recommend it, the **REM** pseudo-op. Suppose a comment was started in the comment field of an instruction, and the comment was too long. Normally, it would be continued by starting the next line with an asterisk, allowing the entire next line to be used for commentary. This makes for difficult reading. If the following line is typed with **REM** in the instruction field the commentary which follows the space after the **REM** will be aligned with comments field of the instruction above it, making the entire commentary a neat, easy to read block. Suppose the instruction in line 7 of example 4-1 required more explanation than just "**CROSS ZERO?**". It could be written this way:

```
BC END THE CARRY BIT GOES ON  
REM WHEN THE RESULT OF THE  
REM SUBTRACTION GOES FROM  
REM POSITIVE TO NEGATIVE
```

which looks like this when listed:

```
>L1 F.
0001 * SQUARE ROOT PROGRAM
0002      ORG      :100      START HERE, BRUNHILDE
0003      LHL D      N      GET NUMBER
0004      LXI      D,1      1ST ODD INTEGER TO D-E
0005      RSC              CLEAR CARRY
0006 LOOP   DSC      D      SUBTRACT ODD INTEGER
0007      BC      END      THE CARRY BIT GOES ON
0008      REM              WHEN THE RESULT OF THE
0009      REM              SUBTRACTION GOES FROM
0010      REM              POSITIVE TO NEGATIVE
0011      INX      D      NO, FORM NEXT ODD
0012      INX      D      INTEGER
0013      BRA      LOOP     AND DO IT AGAIN
0014 END     LRD              DOUBLE SHIFT THE
0015      RER              RESULT. ROOT IN E REG.
0016 HERE   JMP      HERE
0017 N      DBL      65535
0018      END
```

and in the assembly listing looks like this:

PAGE 01

```
001      * SQUARE ROOT PROGRAM
002      ORG      :100      START HERE, BRUNHILDE
003 0100 2A1601      LHL D      N      GET NUMBER
004 0103 110100      LXI      D,1      1ST ODD INTEGER TO D-E
005 0106 B7          RSC              CLEAR CARRY
006 0107 ED52      $ LOOP   DSC      D      SUBTRACT ODD INTEGER
007 0109 3804      $      BC      END      THE CARRY BIT GOES ON
008              WHEN THE RESULT OF THE
009              SUBTRACTION GOES FROM
010              POSITIVE TO NEGATIVE
011 010B 13          INX      D      NO, FORM NEXT ODD
012 010C 13          INX      D      INTEGER
013 010D 18F8      $      BRA      LOOP     AND DO IT AGAIN
014 010F CB3A      $ END     LRD              DOUBLE SHIFT THE
015 0111 CB1B      $      RER              RESULT. ROOT IN E REG.
016 0113 C31301      HERE   JMP      HERE
017 0116 FFFF      N      DBL      65535
018 0118      END
```

```
*****
* S Y M B O L   T A B L E *
*****
```

```
END      010F      HERE      0113      LOOP      0107      N      0116
```

Note that the **REM**'s themselves are blanked out in the assembly listing.



The assembly process itself is initiated by the # command, with a modifier. If #L. is typed a full assembly listing is printed on the system listing device, producing the result just shown. If #E. is typed the program is assembled but only lines which contain errors are listed. This is an extremely useful option, allowing the program to be scanned for misspelled instructions and other typing errors without waiting for an entire assembly listing to be printed. The result of using #E. with the program of example 4-2 in the buffer is:

```

003 2000 3A0620      @      LDA    Q
      S
004 2003 000000      STA    M
      U
006 2007 00000000    FUZZ
      O
007 200B 1800      $      BRA    :3000
      A
009 2010 210000      LXI    H,'ABC'
      L
010 2013 00      MOV    T,M
      R
011 2014 DD7E00      $      LAX    300
      D
013 201A      DATA    @78
      U
014 201A FE00      CPI    ''
      L

```

Only the lines containing errors are listed. The program can then quickly be repaired using D and A commands. A full assembly listing need only be made when all of these clerical errors have been eliminated.

The third assembly mode is selected by typing #O. This is "oh" not zero. This causes an object program to be punched out on the system paper tape punch device, with about six inches of blank tape at either end. The program pauses at both the beginning and end of punching to allow the punch to be turned on and off. This is to accommodate systems in which all of the functions are performed by a single device like a Teletype®.

*Teletype is a registered trademark of Teletype Corporation.*

The tape which is produced by the #O. command is a machine language version of the source program. The contents of this tape can be loaded into computer memory and executed under control of the debugging monitor in appendix C. The loading of the tape and execution of the program under control of the debugging monitor are described in chapter 17.

A fourth assembly mode is entered by the #Z. command. This mode scans for and prints lines which contain Z80 only instructions, i.e., instructions which are not part of the 8080 set. This assembly mode is provided to allow easy scanning for Z80 instructions in programs which must maintain compatibility with the 8080.

A summary of all text editor/assembler commands and means for assembling programs which are too long to fit in memory are given in appendix B.

Finally, the assembler allows each page of the assembly listing to be headed with any text of the programmer's choice. Look at the listing of the debugging monitor in appendix C. The heading:

### ZEDBUG MONITOR

printed on the same line as the page number was specified by a **TITL** pseudo-op. If **TITL** appears anywhere in a source program, the contents of its operand field, up to 36 characters, will be used as a heading for each page. If more than one **TITL** appears in a program only the last one to appear is effective.

Having now completed the discussion of these mechanical details we can proceed to the real business of this book — learning to program the Z80 computer.

## 6 / MOVING DATA WITH Z80 INSTRUCTIONS

“He bought a large map representing the sea  
Without the least vestige of land;  
And the crew were much pleased when they found it to be  
A map they could all understand.”

Lewis Carroll  
*The Hunting of the Snark*

All activity which takes place in a computer revolves around the movement of data, between registers and memory, register to register, or between the computer and the outside world. The Z80 computer provides a number of modes of data movement approached by no other eight bit microcomputer, and it is in these modes that the power of the Z80 lies.

Data may be moved between registers and memory in a variety of ways. The movement of data from memory to a register or register pair is known as *loading*. Movement of data from a register or register pair to memory is known as *storing*. These terms are quite specific in meaning and represent the common practice of the programming craft. They are fundamental and must be memorized before any further progress can be made. When the data movement can occur in either direction the general term *move* is used.

The simplest kind of data movement is direct loading and storing. The Z80's A register can be loaded from memory and stored into memory by means of the **LDA** and **STA** instructions. A simple use of these instructions is shown in example 6-1.

### Example 6-1

Two memory locations have been labeled **DICKY** and

**JAIL.** Use the **LDA** and **STA** instructions to move **DICKY** to **JAIL**.

|              |              |                |                                           |
|--------------|--------------|----------------|-------------------------------------------|
| <i>label</i> | <i>inst.</i> | <i>operand</i> |                                           |
|              | <b>LDA</b>   | <b>DICKY</b>   | contents of <b>DICKY</b> to<br>A register |
|              | <b>STA</b>   | <b>JAIL</b>    | store A register in <b>JAIL</b>           |
|              | ...          |                |                                           |

Register pairs may be loaded and stored directly by the following instructions:

|             |               |
|-------------|---------------|
| <b>LHLD</b> | Load H and L  |
| <b>SHLD</b> | Store H and L |
| <b>LBCD</b> | Load B and C  |
| <b>SBCD</b> | Store B and C |
| <b>LDED</b> | Load D and E  |
| <b>SDED</b> | Store D and E |
| <b>LXD</b>  | Load X        |
| <b>SXD</b>  | Store X       |
| <b>LYD</b>  | Load Y        |
| <b>SYD</b>  | Store Y       |

Another 16 bit register, named SP, can be loaded and stored directly by means of **LSPD** and **SSPD**, but we will come to this later. Just as with **LDA** and **STA** in example 6-1, the register pair loads and stores must specify the operand. A simple exercise with some of these registers is shown in example 6-2.

#### Example 6-2

Use the register pair load and store instructions discussed above to exchange the contents of the BC and DE registers pairs. The idea here is that the contents of BC end up in

DE and vice versa. This is done by using memory locations to hold the registers' contents. Both register pairs are stored, then reloaded with each other's contents. Since register *pairs* are being stored, two words must be allowed in memory for each of the stores. A convenient way to create a two-word space is:

| <i>label</i> | <i>inst.</i> | <i>operand</i> |                                                                                        |
|--------------|--------------|----------------|----------------------------------------------------------------------------------------|
| <b>HENRY</b> | <b>DBL</b>   | <b>0</b>       | reserve 2 words under<br>the name <b>HENRY</b><br>and 2 under the name<br><b>ALICE</b> |
| <b>ALICE</b> | <b>DBL</b>   | <b>0</b>       |                                                                                        |
| ...          |              |                |                                                                                        |

With the temporary storage words defined under the names **HENRY** and **ALICE** we can proceed with the exchange of the register contents.

|             |             |              |                                     |
|-------------|-------------|--------------|-------------------------------------|
| <b>SWAP</b> | <b>SBCD</b> | <b>HENRY</b> | save BC contents<br>in <b>HENRY</b> |
|             | <b>SDED</b> | <b>ALICE</b> | save DE contents<br>in <b>ALICE</b> |
|             | <b>LBCD</b> | <b>ALICE</b> | load BC from <b>ALICE</b>           |
|             | <b>LDED</b> | <b>HENRY</b> | and DE from <b>HENRY</b>            |
| ...         |             |              |                                     |

It needs to be emphasized that this is merely an illustration of the use of these instructions. There is a much better way to accomplish the same thing using the Z80's stack which will be shown in a later example.

Data are also moved in the Z80 by the use of register pair *pointers*. These pointers are similar in function to that discussed for the program counter in chapter 3. Recall that the program counter was said to "point to", i.e., contain the address of, the next instruction to be fetched from memory. A register pair may be used in a similar way



to “point to” a memory location to be accessed. There are a number of modes in which this can be done. The most fundamental involves the use of the HL register pair. If the HL register pair contained the hex number 2AC4, then the instruction:

**MOV   A,M**

will load the contents of memory location 2AC4 into the A register. The other single registers, B, C, H, L, D, and E can also be loaded in this way, i.e.:

**MOV   B,M**  
**MOV   C,M**  
**MOV   H,M**  
**MOV   L,M**  
**MOV   D,M**  
**MOV   E,M**

Note the order of specification of the operands, first the destination, then the source. Note also that the combinations **MOV H,M** and **MOV L,M** are not useful, since they destroy the HL pointer being used to move the data.

Any of the single registers may also be stored into memory by simply reversing the order of the operands, i.e.:

**MOV   M,A**

stores the contents of A into the memory location pointed to by HL, while the combinations:

**MOV   M,B**  
**MOV   M,C**  
**MOV   M,H**  
**MOV   M,L**  
**MOV   M,D**  
**MOV   M,E**

store the contents of the named registers in the memory locations pointed to by HL. Again **MOV M,H** and **MOV M,L** com-

binations are not useful. What must be remembered here is the order of the operands for a **MOV**, first the destination, then the source, i.e.:

**MOV** (to here),(from here)

The combination **MOV M,M** is forbidden and will be flagged by the assembler as an error.

The **MOV** instruction can also be used to transfer data between registers. The operands of the **MOV** are the two registers, e.g.:

**MOV B,L**

transfers the contents of the L register to the B register. Again, the order of operands is destination, then source. These register moves are illustrated in example 6-3.

#### Example 6-3

Using the A register as temporary storage, swap the contents of the L register with those of the C register. The method is the same as that of example 6-2. First we copy one of the registers, say L, into the temporary storage space (A). The second register, C, is then moved to L. Finally, the temporary word in A is moved to C.

| <i>label</i> | <i>inst.</i> | <i>operand</i> |                          |
|--------------|--------------|----------------|--------------------------|
| <b>XCL</b>   | <b>MOV</b>   | <b>A,L</b>     | save L contents in A     |
|              | <b>MOV</b>   | <b>L,C</b>     | transfer C contents to L |
|              | <b>MOV</b>   | <b>C,A</b>     | former L contents to C   |
|              |              | ...            |                          |

the same problem can be solved just as easily another way:

|            |            |                          |
|------------|------------|--------------------------|
| <b>MOV</b> | <b>A,C</b> | save C contents in A     |
| <b>MOV</b> | <b>C,L</b> | transfer L contents to C |
| <b>MOV</b> | <b>L,A</b> | former C contents to L   |
|            |            | ...                      |

In summary, **MOV** may be used to move data between registers, or between registers and memory. The direction of data movement is specified as destination, then source. If either operand of a move is specified as **M**, the data movement takes place between the register named as the other operand and the memory word pointed to by the 16 bit number in **H** and **L**.

Yet another addressing mode allows pointers held in the **BC** and **DE** register pairs to be used in the transfer of data between memory and the **A** register, but only the **A** register. This is done with the **LDAX** and **STAX** instructions. The operand of an **LDAX** or **STAX** consists of a single character, **B** or **D**, specifying either the **BC** or **DE** register pair as the pointer. The use of this mode is shown in example 6-4.

#### Example 6-4

Assume that the **BC** register pair contains the address of (points to) a memory location named **CONGRS** and that the **DE** register pair contains the address of (points to) a memory location named **MONEY**. Use **LDAX**, **STAX** and **MOV** instructions to exchange the contents of **CONGRS** and **MONEY**.

| <i>label</i> | <i>inst.</i> | <i>operand</i> |                                             |
|--------------|--------------|----------------|---------------------------------------------|
| <b>MSWAP</b> | <b>LDAX</b>  | <b>B</b>       | <b>CONGRS</b> to <b>A</b> register          |
|              | <b>MOV</b>   | <b>H,A</b>     | save it in <b>H</b>                         |
|              | <b>LDAX</b>  | <b>D</b>       | <b>MONEY</b> to <b>A</b> register           |
|              | <b>STAX</b>  | <b>B</b>       | store <b>MONEY</b><br>in <b>CONGRS</b>      |
|              | <b>MOV</b>   | <b>A,H</b>     | now retrieve <b>CONGRS</b><br>from <b>H</b> |
|              | <b>STAX</b>  | <b>D</b>       | and store it in <b>MONEY</b>                |
|              |              | ...            |                                             |

The swap can be done in the opposite order, of course, as in example 6-3.

An even more versatile method of data movement is available by using the X and Y registers. Data may be loaded into any single register through a pointer held in X or Y via load instructions of the following form:

**LAX     0**

to load the A register with the contents of the memory location pointed to by X. B would be loaded through the pointer in Y by:

**LBY     0**

We will get to the zeros in the operand fields in a moment. Storing is accomplished by instructions of the form:

**SBX     0**

to store the B register through X, and:

**SLY     0**

to store L in the location pointed to by Y. To avoid the awkwardness of the expressions "pointed to" and "via the pointer contained in" we will hereafter use the nomenclature *through*. Thus the instruction:

**LCX     0**

loads the C register *through* X, and:

**SEY     0**

stores the E register *through* Y. It must be clear, however, that the data do not actually travel through the pointer register.

The instructions which load and store through X and Y illustrated above all had operand fields of zero. The number in the operand field is known as a *displacement*. Instructions like **LAX** and **SLY** which reference memory through the X and Y registers are said to be *indexed* instructions, and the X and Y registers themselves are re-

ferred to as *index registers*. The displacement of an indexed instruction is added to the contents of the index register before memory is actually accessed. If the displacement of **LBY** were a 3, the effect would be to load the B register with the contents of the memory word whose address was 3 greater than that pointed to by Y. This does *not* change the contents of Y. The computation of the final address of the memory word involved is done internally and Y (or X) is not changed. This has a great many applications and its judicious use can simplify a number of programming tasks.

The displacement of an indexed instruction need not be a numerical constant. Suppose we were working on a hardware inventory problem in which successive memory locations contained the number of nuts, bolts, nails, screws and pins. Using the **EQU** pseudo-op a table of equivalences can be created:

|               |            |          |
|---------------|------------|----------|
| <b>NUTS</b>   | <b>EQU</b> | <b>0</b> |
| <b>BOLTS</b>  | <b>EQU</b> | <b>1</b> |
| <b>NAILS</b>  | <b>EQU</b> | <b>2</b> |
| <b>SCREWS</b> | <b>EQU</b> | <b>3</b> |
| <b>PINS</b>   | <b>EQU</b> | <b>4</b> |

These names can now be used as equivalent to the numbers they represent. If the Y register were loaded to point to the words containing the count of nuts, the instruction:

**LBY     NUTS**

is entirely equivalent to writing:

**LBY     0**

and the first instruction is more meaningful in terms of the problem to be solved. Further, the instruction:

**LBY     SCREWS**

is the same as **LBY 3**, but the programmer is relieved of the task of remembering where the count of screws is stored every time he wants to work with it. Even more important, such a program will be



accessing these locations in many places. If the locations ever change, the program can only be updated by changing every reference to the numerically defined displacements. If the displacements are defined in terms of symbols as shown, however, only the small table of EQUs needs to be changed, and the assembler will automatically correct every reference when the program is reassembled. This illustrates a very important programming tactic. Wherever possible, *make the programming system do the work for you.*

Discussion up until now has centered around pointers contained in registers, but how the pointers came to be there has been left unexplained. A pointer is a 16 bit quantity. The assembler contains facilities for the creation of double length constants in memory. The source statement:

```
M1000 DBL -1000
```

will result in two words being generated which contain the binary equivalent of the decimal number -1000. This number,  $111110000011000_2$ , is stored in two consecutive words, the low significance half of the number occupying the word with the lower memory address. Note that this order is different from that used by some other microprocessors. Since a label symbol has a numeric value assigned by the assembler this value may be considered to be a 16 bit numeric constant. A constant of this sort is said to be an *address constant*. The 16 bit address constants for a pair of locations named **TRISTN** and **ISOLDE** can be created by:

```
TPTR DBL TRISTN  
IPTR DBL ISOLDE
```

An address constant can be loaded directly into a register pair, e.g.:

```
LHLD TPTR
```

would load the address of **TRISTN** into the HL register pair. Thereafter the contents of **TRISTN** could be moved to the A register by:

```
MOV A,M
```

While it is sometimes necessary to use address constants stored in memory, it is usually possible to get the address constant into the register pair another, more economical way. This involves the use of a new addressing mode, the so-called *immediate mode*. The operand of an immediate instruction is contained within the instruction itself. This makes an ideal way to generate program constants without using extra memory to hold them. The A register can be loaded with the number -35 by:

**MVI     A,-35**

or the C register with the ASCII code for the letter P by:

**MVI     C,'P'**

the alphabetic nature of the constant being indicated by the apostrophes. Register pairs can be loaded with 16 bit constants as shown in line 8 of example 4-2. To load a register pair with a constant the **LXI** instruction is used, i.e.:

|            |                |                                           |
|------------|----------------|-------------------------------------------|
| <b>LXI</b> | <b>H,1000</b>  | loads H and L with the number 1000        |
| <b>LXI</b> | <b>B,0</b>     | clears the BC register pair               |
| <b>LXI</b> | <b>D,-1</b>    | loads D and E with $FFFF_{16}$            |
| <b>LXI</b> | <b>X,:3F7B</b> | loads X with $3F7B_{16}$                  |
| <b>LXI</b> | <b>Y,'OK'</b>  | loads Y with the ASCII characters O and K |

Since a label has a numeric value, we can generate immediate address constants the same way, so that instead of creating an address constant in memory as we did above, the constant can become part of the instruction which loads it. In this way:

**LXI     H,TRISTN**

loads the address of **TRISTN** into the HL register pair. This saves the two words required for the constant **TPTR**. The use of pointers loaded in this way is shown in example 6-5.

### Example 6-5

Using pointers held in the HL and DE register pairs move one word of data from **ALPHA** to **OMEGA**.

| <i>label</i> | <i>inst.</i> | <i>operand</i> |                             |
|--------------|--------------|----------------|-----------------------------|
|              | <b>LXI</b>   | <b>H,ALPHA</b> | address of <b>ALPHA</b>     |
|              |              |                | to HL                       |
|              | <b>LXI</b>   | <b>D,OMEGA</b> | address of <b>OMEGA</b>     |
|              |              |                | to DE                       |
|              | <b>MOV</b>   | <b>A,M</b>     | contents of <b>ALPHA</b> to |
|              |              |                | A register                  |
|              | <b>STAX</b>  | <b>D</b>       | store in <b>OMEGA</b>       |
|              | <b>...</b>   |                |                             |

The method of example 6-5 is too longwinded for moving a single word of data, of course. If only one word were to be moved it could be done more economically and faster with:

```

LDA    ALPHA
STA    OMEGA

```

The point of moving the data by the method of example 6-5 was to introduce the possibility of moving an entire block of data at a time. To do this the pointers held in HL and DE need to be manipulated. Suppose a five word block of data beginning at **METS** were to be moved to a five word block beginning at **PHILS**. It could be accomplished by a sequence of **LDA** and **STA** instructions, i.e.:

```

LDA    METS
STA    PHILS
LDA    METS+1
STA    PHILS+1
LDA    METS+2
STA    PHILS+2
...

```

and so forth. A better way is to move the data by means of pointers which can be changed to point to different memory addresses. The Z80 has instructions which can *increment*, i.e., increase by one, and *decrement*, i.e., decrease by one, each of the pointer register pairs. These instructions are:

|            |           |                         |
|------------|-----------|-------------------------|
| <b>INX</b> | <b>B</b>  | increment BC pair       |
| <b>DCX</b> | <b>B</b>  | decrement BC pair       |
| <b>INX</b> | <b>H</b>  | increment HL pair       |
| <b>DCX</b> | <b>H</b>  | decrement HL pair       |
| <b>INX</b> | <b>D</b>  | increment DE pair       |
| <b>DCX</b> | <b>D</b>  | decrement DE pair       |
| <b>INX</b> | <b>X</b>  | increment X register    |
| <b>DCX</b> | <b>X</b>  | decrement X register    |
| <b>INX</b> | <b>Y</b>  | increment Y register    |
| <b>DCX</b> | <b>Y</b>  | decrement Y register    |
| <b>INX</b> | <b>SP</b> | increment stack pointer |
| <b>DCX</b> | <b>SP</b> | decrement stack pointer |

We will have more to say about these last two instructions a bit later. Before we can move a block of data with pointers, there remains one problem to be solved; how to control the size of the block which is moved. In this case the size of the block is five, so we want to create a count of five somewhere and decrement it each time a word is moved, stopping when the count becomes zero. To do this we could use the **INR** or **DCR** instructions. **INR** addresses a single eight bit register, i.e.:

|            |          |                                                                |
|------------|----------|----------------------------------------------------------------|
| <b>INR</b> | <b>A</b> | adds one to the number<br>which is in the A<br>register, while |
| <b>INR</b> | <b>C</b> | does the same thing<br>for C                                   |

The **DCR** instruction decrements an eight bit register, i.e.:

|            |          |                                                                     |
|------------|----------|---------------------------------------------------------------------|
| <b>DCR</b> | <b>H</b> | subtracts one from the<br>value in the eight bit H<br>register, and |
|------------|----------|---------------------------------------------------------------------|

**DCR     D**does the same thing  
to D

At this point we must know something about the F register which has remained unexplained until now. The execution of **INR** or **DCR** instructions set the flags of the F register to reflect the result of the incrementation or decrementation, specifically, the *zero flag* or *Z bit* is set if the result of an **INR** or **DCR** is zero. The only problem which remains is to use this flag setting to stop the movement of the data. The flags are tested and the flow of the program changed by means of *jump* or *branch* instructions. The difference between a jump and a branch is not important just now. We will use a branch instruction here. The point is that a test is made of a bit in the F register. If the bit is a one the sequential execution of instructions is altered by loading a new value into the program counter, this value being specified by the operand of the jump or branch. The branch instruction:

**BZ        AGAIN**

means that if the result of the last operation (**INR** or **DCR** here) was a zero take the next instruction from the memory location whose name is **AGAIN**. If the result was not zero, simply take the instruction following the **BZ**.

The **BNZ** instruction tests on the opposite condition. If the result of the last operation was zero the instruction after the **BNZ** is taken. If the result was not zero the next instruction is taken from the location named in the operand of the **BNZ**.

Other result conditions can also be tested by branches. If the last arithmetic operation resulted in a carry out of the high bit the carry bit in the F register control may be transferred to a new location by means of the **BC** or **BNC** instructions. The **BC** branches if the carry bit is a one. If carry is zero the next instruction which follows the **BC** is taken. **BNC**, meaning branch if no carry, has just the opposite effect. The branch is taken if carry is a zero. If carry is a one the instruction which follows the **BNC** is executed.

If no particular condition is to be tested control can be transferred to a new program location by a simple **BRA** (branch) instruction, i.e.:



**BRA NEWLOC**

transfers control to **NEWLOC** under all circumstances.

We can now put all of this together to move the five word block from **METS** to **PHILS**. This move is shown in example 6-6.

**Example 6-6**

Using pointers held in HL and in DE, move the five word block of data which begins at **METS** to the five word block which begins at **PHILS** under control of a count held in the B register.

| <i>label</i>  | <i>inst.</i> | <i>operand</i> |                                                                                                              |
|---------------|--------------|----------------|--------------------------------------------------------------------------------------------------------------|
| <b>BLMOVE</b> | <b>MVI</b>   | <b>B,5</b>     | load the constant 5 into B                                                                                   |
|               | <b>LXI</b>   | <b>H,METS</b>  | address of <b>METS</b> to HL                                                                                 |
|               | <b>LXI</b>   | <b>D,PHILS</b> | address of <b>PHILS</b> to DE                                                                                |
| <b>AGAIN</b>  | <b>MOV</b>   | <b>A,M</b>     | move word from <b>METS</b> block to A register                                                               |
|               | <b>STAX</b>  | <b>D</b>       | and then to <b>PHILS</b> block                                                                               |
|               | <b>INX</b>   | <b>H</b>       | increment <b>METS</b> address pointer in HL. This makes HL point to the next word in the <b>METS</b> block   |
|               | <b>INX</b>   | <b>D</b>       | increment <b>PHILS</b> address pointer in DE. This makes DE point to the next word in the <b>PHILS</b> block |
|               | <b>DCR</b>   | <b>B</b>       | decrement the count of the B register. This sets the Z bit of the F register so that it can be               |

**BNZ AGAIN**

...

tested by the **BNZ** if the result of the **DCR** in the line above was a zero, control will pass to the instruction which follows the **BNZ**. If the result was not zero, the next instruction will be taken from location **AGAIN** program continues here

It is of crucial importance that the reader understand completely what went on in example 6-6. Example 6-6 illustrates the all-important principle of *looping*, the repetitive execution of a section of a program under control of a count. If this example is not understood now, very little of the remainder of this book will make sense. If you are the least bit fuzzy about example 6-6, go back over it until you do understand it. The importance of the *looping* principle cannot be overstressed.

Example 6-6 showed the movement of data under control of a simple count held in B. The example can be shortened by using a Z80 facility which combines two 8080 instructions into one, the **DBNZ** instruction. **DBNZ** combines the **DCR** and **BNZ** instructions. This single instruction decrements the B register and performs the branch test. The program in example 6-6 would read:

```
BLMOVE  MVI  B,5
        LXI  H,METS
        LXI  D,PHILS
AGAIN    MOV  A,M
        STAX D
        INX  H
        INX  D
        DBNZ AGAIN
        ...
```

While we have not yet discussed the amount of memory required by the various instructions, use of the **DBNZ** shortens the program in example 6-6 by one word. **DCR** and **BNZ** together require three words, **BNZ** being two of the three. **DBNZ** requires only two words.

It is not always possible to use a branch instruction to transfer control. The reason for this is that the range over which a branch can transfer control is limited. The point to which control is transferred by a branch must be within the limits  $+127$  to  $-128$  of the program counter value when the branch is executed. The point to which control is transferred by a branch is computed by the Z80 by adding the second word of the branch instruction to the program counter. Recall now the earlier emphasis on the value of the program counter. *The program counter always points to the instruction after the one being executed.* Since branch instructions are two words long this means that the location to which control is transferred must be in the range  $+129$  to  $-126$  of the location of the first word of the branch instruction. This relatively short range does not allow a branch to reach all parts of memory. For this reason the Z80 also has *jump* instructions. These instructions are three words long, compared to two for a branch, but a jump can reach anywhere in memory. The target address of a jump is not computed from the program counter but is contained directly in the last two words of the instruction. In addition, jumps may test on more conditions than branches, so they offer more versatility. The jump instructions are:

|            |                                             |
|------------|---------------------------------------------|
| <b>JZ</b>  | jump if zero result                         |
| <b>JNZ</b> | jump if not zero result                     |
| <b>JC</b>  | jump if carry is a one                      |
| <b>JNC</b> | jump if carry is a zero                     |
| <b>JV</b>  | jump if an arithmetic overflow has occurred |
| <b>JNV</b> | jump if no arithmetic overflow has occurred |
| <b>JP</b>  | jump if result was positive                 |
| <b>JM</b>  | jump if result was negative                 |
| <b>JMP</b> | jump unconditionally                        |

The exact meanings of all these conditions are not important just yet. They will become clear as they are used in the examples in fur-

ther chapters of this book. Two other jumps, **JPE** and **JPO**, perform the same test as **JNV** and **JV** respectively. The mnemonics mean jump if parity even and jump if parity odd. Parity refers to the evenness or oddness of the number of one bits in an eight bit group. This is of interest to communications specialists but will not be of any concern to us in this book. Note, however, that **JNV** and **JV** are not flagged by the assembler as Z80 only instructions (\$), being equivalent to the 8080's **JPE** and **JPO**.

If either a jump or a branch can be used in a situation, use the branch. If you attempt to branch too far the assembler will let you know when you execute the error scan (#E.) by flagging the line with an "A", meaning addressing error. The equivalent jump can then be easily substituted.

Going back now to the movement of data, the Z80 has facilities for moving blocks of data available on no other current microcomputer. It is possible to move an entire block of data with a single Z80 instruction, and thus to perform the function shown in example 6-6 with only four instructions instead of the nine used there. Using pointers held in HL and DE and a 16 bit count held in BC, up to 65,536 words of memory may be moved by a single instruction. The mnemonic for this powerful instruction is **MW+R**, the individual characters meaning move word increment and repeat. When **MW+R** is executed a number of things happen. First the contents of the word pointed to by HL are fetched from memory and stored into the memory location pointed to by DE. Both pointers, HL and DE are then incremented. The BC pair is then decremented. If the result of decrementing BC is not zero, the program counter is "backed up" and the instruction is repeated, but with the changed values of the pointers in HL and DE. Using **MW+R** the function of the program in example 6-6 is shown in example 6-7.

#### Example 6-7

Using the block move instruction **MW+R**, move the five word block which begins at **METS** to the five word block which begins at **PHILS**.

| <i>label</i>  | <i>inst.</i> | <i>operand</i> |                                                                                                                                                       |
|---------------|--------------|----------------|-------------------------------------------------------------------------------------------------------------------------------------------------------|
| <b>BLMOVE</b> | <b>LXI</b>   | <b>B,5</b>     | load the BC pair with the number 5. Note that this load differs from the one in example 6-6 in that the constant 5 occupies the entire 16 bit BC pair |
|               | <b>LXI</b>   | <b>H,METS</b>  | <b>METS</b> pointer to HL as before                                                                                                                   |
|               | <b>LXI</b>   | <b>D,PHILS</b> | <b>PHILS</b> pointer to DE as before                                                                                                                  |
|               | <b>MW+R</b>  |                | move entire block with one instruction                                                                                                                |
|               | <b>...</b>   |                |                                                                                                                                                       |

An interesting application of this instruction is in filling a block of memory locations with some constant value. To do this the value to fill the block is stored into the first word. The length of the block minus one is then used as the count in BC. HL is made to point to the first word of the block, the one already containing the desired value. DE is then made to point to the word immediately following. **MW+R** is then executed, with the following effect. The first word of the block is transferred to the second. This is simply a copy of the first word into the second. HL and DE are then incremented so that they point to the second and third words of the block respectively. BC is decremented and if the result of the decrementation is not zero the process is repeated, moving the second word of the block to the third, the remainder of the block being filled by successive iterations of this process. A program to do this is shown in example 6-8.

#### Example 6-8

Using the **MW+R** instruction, fill the block beginning at **HERE** and ending at **THERE** with the binary value of the



ASCII character "\$". **THERE** has a higher memory address than **HERE**.

| <i>label</i> | <i>inst.</i> | <i>operand</i>      |
|--------------|--------------|---------------------|
| <b>FILL</b>  | <b>LXI</b>   | <b>B,THERE-HERE</b> |

Since both **HERE** and **THERE** have been assigned values by the assembler, these values can be subtracted to find the length of the block to be filled. The value of **THERE-HERE** is actually one less than the length of the block

|  |            |               |
|--|------------|---------------|
|  | <b>LXI</b> | <b>H,HERE</b> |
|--|------------|---------------|

address of beginning of block to HL

|  |            |                   |
|--|------------|-------------------|
|  | <b>LXI</b> | <b>D,HERE + 1</b> |
|--|------------|-------------------|

address of second word of block to DE. HL now points to the first word of the block and DE to the second. BC contains the number of copying operations that have to be done, i.e., one less than the block length.

|  |            |               |
|--|------------|---------------|
|  | <b>MVI</b> | <b>A,'\$'</b> |
|--|------------|---------------|

|  |            |            |
|--|------------|------------|
|  | <b>MOV</b> | <b>M,A</b> |
|--|------------|------------|

now get the fill value and put it into the first word of the block

|  |             |
|--|-------------|
|  | <b>MW+R</b> |
|--|-------------|

copy to rest of block

|  |            |
|--|------------|
|  | <b>...</b> |
|--|------------|

This ability to move entire blocks of data is extremely useful in writing programs such as text editors. The text editor used by the as-

sembler in appendix B does not use **MW+R** because it was required that the assembler run entirely within the 8080 subset of the Z80, but it could have been used to great advantage. Consider the situation in which a line or block of lines has been deleted from the text buffer. It is required that all text above this, i.e., line numbers greater than those that have been deleted be moved down to fill the gap created by the deletion. If the length of remaining text above the deleted portion is in BC, the address of the first character to be moved down in HL and the address of the first word of the gap in DE, then the single instruction **MW+R** fills the gap.

A similar instruction, **MW-R**, meaning move word decrement and repeat, moves a block of data just like **MW+R**, but the HL and DE pointers are decremented rather than incremented. This is useful when a gap must be created, as when a line is added to a text buffer. All of the text above the point at which the material is to be added must be moved up by an amount equal to the length of the text to be added. In this case the count of characters to be moved is the length of the text beyond the addition point. HL contains the address of the last text character and DE contains the address to which this character is to be moved, i.e., the address of the last character plus the length of the gap which must be created for the new text. When the pointers and count have been set up this way, the single instruction **MW-R** creates the necessary gap. The new text can then be copied into the gap with the **MW+R** instruction.

The **MW+R** and **MW-R** instructions have non repeating forms **MW+** and **MW-**. These instructions perform the same move function as their repeating counterparts except that only one word is moved per execution of the instruction. HL and DE are incremented or decremented and BC is decremented. **MW+** and **MW-** turn the overflow bit on, i.e. a one, if BC is nonzero *after* execution. If BC is zero after executing **MW+** or **MW-** the overflow bit is cleared, i.e., made zero.

The transfer of data by **MW+**, **MW+R**, **MW-**, or **MW-R** does not involve the Z80's A register. A constant may be held in A during the transfer.

An interesting application of the Z80 block move instructions is to data structures known as *stacks*. One type of stack behaves something like a section of railroad track which has room for some exact number of cars. If a car is pushed into one end of the stack another

car is pushed out of the other end. This kind of structure is known as a FIFO stack, meaning **First In First Out**. Such a stack can be used to hold a "model" of a piece of track. An application of this is shown in example 6-9.

#### Example 6-9

A section of railroad track is to be modeled in the computer. This section of track will hold exactly 36 cars. Each of the 36 positions in the model of the track section is to hold the identifying number of the car in that relative position on the track. A new car is "pushed" into the model by moving the entire block of identifiers up to make room for the new one. This forces the oldest identifier out the top. Beginning with the new identifier in the A register, enter it into the block and finish with the identifier which has been pushed out in A.

This brings us to the necessity of knowing how to make the assembler reserve a block of memory. This is done by the **RES** and **BSS** pseudo-ops. **RES**, meaning **REServe**, and **BSS**, meaning **Block Started by Symbol**, are different ways of saying the same thing. They both direct the assembler to assign a block of memory whose length is given in the operand of the **RES** or **BSS**. A block of 36 locations can be reserved under the name **TRACK** by:

```
TRACK RES 36
```

or

```
TRACK BSS 36
```

The symbol **TRACK** refers to the first locations of the block, i.e., the one with the lowest memory address. The last location of the block is **TRACK + 35**. **RES** and **BSS** will accept a second operand, if desired. If the second op-

erand is given, it is taken to mean the value with which the block is to be filled, e.g.:

**TRACK BSS 36,-3**

will result in the entire block being filled with  $11111101_2$  when the program is loaded for execution, while:

**TRACK RES 36,'G'**

will fill the block with  $11000111_2$ , the ASCII code for the letter G.

There are several ways to solve the problem posed in this example. One would be to save the last word, the one to be pushed out, then move the block, then insert the new word into the lowest block address, **TRACK**. This operation looks like this:

| <i>label</i> | <i>inst.</i> | <i>operand</i>    |                          |
|--------------|--------------|-------------------|--------------------------|
|              | <b>STA</b>   | <b>NEW</b>        | save new value           |
|              | <b>LXI</b>   | <b>B,35</b>       | move count to B. Note    |
|              |              |                   | that though there are 36 |
|              |              |                   | words in the block there |
|              |              |                   | will be only 35 moves    |
|              | <b>LXI</b>   | <b>D,TRACK+35</b> | "to" address to DE       |
|              | <b>LXI</b>   | <b>H,TRACK+34</b> | "from" address to HL     |
|              | <b>LDA</b>   | <b>TRACK+35</b>   | save old value to be     |
|              |              |                   | pushed out               |
|              | <b>STA</b>   | <b>OLD</b>        | ...                      |
|              | <b>MW-R</b>  |                   | move the block up        |
|              | <b>LDA</b>   | <b>NEW</b>        | retrieve new value       |
|              | <b>STA</b>   | <b>TRACK</b>      | place at bottom of stack |
|              | <b>LDA</b>   | <b>OLD</b>        | and retrieve the         |
|              |              |                   | "pushed out" value       |
|              |              |                   | program continues        |
|              |              |                   |                          |

**NEW** and **OLD** are presumed here to have been defined elsewhere in the program. This will work, but is somewhat clumsy, and there is a better way. Suppose the locations **NEW** and **OLD** in example 6-9 had been defined to be *part* of a block named **TRACK**, like this:

```

NEW      ***
TRACK RES 36
OLD      ***

```

The problem then becomes one of managing a 38 word block beginning with **NEW** and ending with **OLD**. The new entry is simply placed in **NEW** and the entire block including **NEW** and **OLD** is moved. This makes the solution to the problem shorter, as shown in example 6-10.

#### Example 6-10

Using the method just described, solve the problem posed in example 6-9.

| <i>label</i> | <i>inst.</i> | <i>operand</i>      |                                                                                                |
|--------------|--------------|---------------------|------------------------------------------------------------------------------------------------|
|              | <b>STA</b>   | <b>NEW</b>          | save new value. This places <b>NEW</b> immediately <i>below</i> the first word of <b>TRACK</b> |
|              | <b>LXI</b>   | <b>B,37</b>         | the count is now 37 because the block has increased in size to 38                              |
|              | <b>LXI</b>   | <b>D,OLD</b>        | address of the last word of the 38 word block as "to" address in DE                            |
|              | <b>LXI</b>   | <b>H,TRACK + 35</b> | address of last word of <b>TRACK</b> as "from" address                                         |
|              | <b>MW-R</b>  |                     | move the block                                                                                 |
|              | <b>LDA</b>   | <b>OLD</b>          | get the value pushed out                                                                       |
|              | <b>...</b>   |                     |                                                                                                |



Examples 6-9 and 6-10 illustrate something important about the programming craft in general, namely, that there is no single *correct* solution to most problems. There are a variety of methods by which most problems can be solved and the method of attack will vary with the situation. Quite often the solution of a problem involves a trade of memory space for speed of execution or vice versa. Since relatively few computer applications demand great speed the usual practice is to write programs in such a way as to conserve memory use. While memory is not as expensive as it was even a short time ago, it is still usually the most costly single item in the computer. For programs which will run in a large development system this makes no difference, but if the computer is being used as a controller in a manufactured item the extra memory burden caused by sloppy programming practices represents an extra fixed cost in every copy of the manufactured item. The emphasis in this book will therefore usually be on economy of memory use.

## 7 / ARITHMETIC AND LOGICAL OPERATIONS ON THE Z80

“D’ye think th’ colledges has much to do with th’ progress iv th’ wurruld?” asked Mr. Hennessy. “D’ye think,” said Mr. Dooley, “’tis th’ mill that makes the wather run?”

*Mr. Dooley’s Opinions*  
Finley Peter Dunne

The principal arithmetic register of the Z80, and the only register in which logical operations can be performed is the A register. Some limited 16 bit arithmetic can be performed in register pairs, but other than this all arithmetic and logical operations are performed either between A and another 8 bit register or between A and a memory word pointed to by a register pointer.

Addition, subtraction and the logical operations result in the bits of the F register being set to reflect the type of result generated by the operation. The bits and their meanings are:

- S — Sign flag. This flag is a copy of the highest bit (sign bit) of the accumulator. It is set to one if the result was negative and zero if the result was positive. The state of the S bit can be tested and program flow altered by the **JP**, jump if plus, and **JM**, jump if minus, instructions. Two things should be noted here. First, a zero result is considered to be positive, and second, there are no branch (relative jump) instructions capable of testing for plus and minus. The longer jump form must therefore be used.
- Z - Zero flag. This flag is set to one if the result was a zero and to zero if the result was nonzero. If this inversion seems confusing, it need not worry you. The jump and branch instruction test the proper sense of the result, i.e., **BZ** and **JZ** transfer control if the result was zero and

**BNZ** and **JNZ** transfer control if the result was nonzero. The programmer rarely has to deal with the Z flag itself.

- A — Auxiliary carry flag. This bit detects the occurrence of a carry or borrow between the low four bits of A and the high four bits during an add or subtract operation. It cannot be tested by jump or branch instructions. It is used by the **DAA** instruction to rectify the sum or difference of BCD numbers. This operation will be covered in chapter 11.
- V — Overflow/Parity flag. This flag is set to one if the result of an addition or subtraction is too large to be held in the allotted space. It is also used as an indicator for certain other conditions, e.g., that the execution of an **MW+**, **MW-**, **CP+**, or **CP-** instruction has caused the count in the BC to become zero. It is used also to indicate the evenness or oddness of the number of one bits in a result. It can be tested by the **JV**, **JNV**, **JPE** and **JPO** instructions.
- N — Add/Subtract flag. This flag is set to zero by add instructions and to one by subtract instructions. It is used by the **DAA** instruction in conjunction with auxiliary carry to adjust sums or differences of BCD numbers. It cannot be tested by jump or branch instructions.
- C — Carry flag. This bit is set to one if the addition of two numbers, considered as unsigned magnitudes, results in a carry out of the high bit of the sum. It is also set to one if, in the subtraction of two unsigned quantities, the subtrahend is greater than the minuend. The carry bit thus functions as a one bit extension of the generated sum or difference. It is tested by the **JC**, **JNC**, **BC** and **BNC** instructions.

Simple addition to the A register is done in the Z80 by the **ADD** instruction. Addition of another register is done by specifying the register as the operand, i.e.:

**ADD E**

adds the contents of the E register to those of A, leaving the sum in A and the flags set as described above. Addition of the contents of a memory word through a pointer held in HL is done by:

**ADD M**

If the pointer is held in X or Y the form is slightly different. The pointer register identification is part of the mnemonic and the operand specifies the displacement, as for the load and store through X or Y described in chapter 6. Addition through X is done by:

**ADDX 0**

and through Y by:

**ADDY 0**

As with the loads and stores, the displacement need not be zero but may specify any value from +127 through -128. This use of the displacement is shown in example 7-1.

**Example 7-1**

A three word block of numbers is in memory beginning at location **TRIO**. Using a pointer held in X, find the sum of the three numbers.

| <i>label</i> | <i>inst.</i> | <i>operand</i> |                                                         |
|--------------|--------------|----------------|---------------------------------------------------------|
|              | <b>LXI</b>   | <b>X,TRIO</b>  | address of first word of <b>TRIO</b> to X.              |
|              |              |                | X now points to the word with the lowest memory address |
|              | <b>LAX</b>   | <b>0</b>       | load first word into A through X                        |

|             |          |                 |
|-------------|----------|-----------------|
| <b>ADDX</b> | <b>1</b> | add second word |
| <b>ADDX</b> | <b>2</b> | add third word  |
| ...         |          |                 |

Very few real world problems can be solved, however, with addition which is limited to eight bits. For this reason the Z80 contains facilities for the performance of arithmetic on numbers which are more than one word long. Such arithmetic is known as *multiple precision arithmetic*, the case in which the number is two words long being known as *double precision arithmetic*. We will cover the double precision case here. If double precision is understood there is no problem in extending it to cover arithmetic of any required length.

Consider the following double precision binary numbers:

$$\begin{array}{ll} 00000000 & 00000011_2 = +3_{10} \\ 11111111 & 11111111_2 = -1_{10} \end{array}$$

the left or most significant part of these numbers is known as the *high order*, the least significant part being known as the *low order*. Both numbers fit the requirement that nonsignificant leading digits are filled with copies of the sign bit. The addition of these numbers begins with the addition of the low order parts, i.e.:

$$\begin{array}{rcl} & 00000011 & \text{low order augend} \\ + & \underline{11111111} & \text{low order addend} \\ 1 & 00000010 & \text{low order sum} \end{array}$$

this result being the low order sum. The detached one at the left of the low order sum represents the carry which occurred out of the leftmost bit of the sum. The high order sum is formed by adding the high order addend and augend, *and the carry from the low order*, i.e.:

$$\begin{array}{rcl} & 00000000 & \text{high order augend} \\ + & 11111111 & \text{high order addend} \\ + & \underline{1} & \text{carry from low order} \\ 1 & 00000000 & \text{high order sum} \end{array}$$



The final sum is 00000000 00000010<sub>2</sub> = 2<sub>10</sub>. Again, the detached one at the left is the carry out of the leftmost bit of the sum. If the addition were to be done in length greater than double this carry would be added to the third order, the carry from the third order added to the fourth and so on.

The normal **ADD** instruction of the Z80 ignores the previous state of carry. If the **ADC** instruction is used, however, the previous carry is added along with the addend to the augend. In this way carries from lower orders can be propagated to higher orders of a multiple precision sum. Addition with carry of another eight bit register is done, e.g.:

**ADC    D**

which adds the contents of the D register plus previous carry to A. Addition of a memory word to A through a pointer held in HL is done by:

**ADC    M**

and through X or Y:

**ADCX   d**

or

**ADCY   d**

the small d's representing the displacements of the indexed instructions. A double precision addition is shown in example 7-2.

#### Example 7-2

A pair of double precision numbers is BC and DE. Find the sum of the numbers, leaving this result in DE.

*label    inst.    operand*

|            |            |                         |
|------------|------------|-------------------------|
| <b>MOV</b> | <b>A,E</b> | low augend to A         |
| <b>ADD</b> | <b>C</b>   | form low sum            |
| <b>MOV</b> | <b>E,A</b> | save low order sum      |
| <b>MOV</b> | <b>A,D</b> | high augend to A        |
| <b>ADC</b> | <b>B</b>   | add high orders plus    |
|            |            | carry from low order to |
|            |            | form high sum           |
| <b>MOV</b> | <b>D,A</b> | save high sum in D      |
| <b>...</b> |            |                         |

Double precision addition of numbers in memory is illustrated in example 7-3.

### Example 7-3

Double precision numbers are in memory locations named **LUKE** and **KENOBI**. Find the double precision sum and leave it in memory at **TARKIN**.

| <i>label</i> | <i>inst.</i> | <i>operand</i>  |                              |
|--------------|--------------|-----------------|------------------------------|
|              | <b>LXI</b>   | <b>B,LUKE</b>   | address of <b>LUKE</b> to BC |
|              | <b>LXI</b>   | <b>H,KENOBI</b> | address of <b>KENOBI</b>     |
|              |              |                 | to HL                        |
|              | <b>LXI</b>   | <b>D,TARKIN</b> | address of <b>TARKIN</b>     |
|              |              |                 | to DE                        |
|              | <b>LDAX</b>  | <b>B</b>        | low order <b>LUKE</b> to A   |
|              | <b>ADD</b>   | <b>M</b>        | add low order <b>KENOBI</b>  |
|              | <b>STAX</b>  | <b>D</b>        | save low order sum in        |
|              |              |                 | <b>TARKIN</b>                |
|              | <b>INX</b>   | <b>B</b>        | increment <b>LUKE</b>        |
|              |              |                 | pointer                      |
|              | <b>INX</b>   | <b>H</b>        | increment <b>KENOBI</b>      |
|              |              |                 | pointer                      |
|              | <b>INX</b>   | <b>D</b>        | increment <b>TARKIN</b>      |

|             |          |                             |
|-------------|----------|-----------------------------|
| <b>LDAX</b> | <b>B</b> | pointer                     |
| <b>ADC</b>  | <b>M</b> | high order <b>LUKE</b> to A |
|             |          | form high order sum         |
|             |          | including carry from        |
|             |          | low order                   |
| <b>STAX</b> | <b>D</b> | save high order sum in      |
|             |          | <b>TARKIN + 1</b>           |
| <b>...</b>  |          |                             |

Subtraction is accomplished in a similar way. Carry in this case acts as a borrow, the borrow being propagated to the higher order. Simple subtraction is done by the **SUB** instruction, while subtraction including previous borrow is done by the **SBB** instruction. A double precision subtraction is shown in example 7-4.

#### Example 7-4

In a numerical differencing problem a pair of double precision numbers is left in the four memory words which begin at location **N**. Find the difference of these two numbers, subtracting the one with the higher memory address from the one with the lower memory address. Leave the difference in **DIFF**.

| <i>label</i> | <i>inst.</i> | <i>operand</i> |                                |
|--------------|--------------|----------------|--------------------------------|
|              | <b>LXI</b>   | <b>D,N</b>     | address of lower number to DE  |
|              | <b>LXI</b>   | <b>H,N+2</b>   | address of higher number to HL |
|              | <b>LDAX</b>  | <b>D</b>       | get low order minuend          |
|              | <b>SUB</b>   | <b>M</b>       | form low order difference      |
|              | <b>STA</b>   | <b>DIFF</b>    | save low order difference      |
|              | <b>INX</b>   | <b>D</b>       | increment minuend              |

|             |               |                                            |
|-------------|---------------|--------------------------------------------|
| <b>INX</b>  | <b>H</b>      | pointer<br>increment subtrahend<br>pointer |
| <b>LDAX</b> | <b>D</b>      | get high minuend                           |
| <b>SBB</b>  | <b>M</b>      | form high order<br>difference              |
| <b>STA</b>  | <b>DIFF+1</b> | save high order<br>difference              |
| <b>...</b>  |               |                                            |

Before leaving examples 7-2, 7-3 and 7-4, it should be noted that though these examples required no test for overflow, the proper place for such a test is immediately after the high order sum or difference is formed. Overflow of any of the low order sums or differences is meaningless.

Like the data movement instructions of the last chapter the arithmetic instructions have immediate counterparts, forms which contain the constant to be added as part of the instruction. These are:

|            |          |                                      |
|------------|----------|--------------------------------------|
| <b>ADI</b> | <b>n</b> | add immediate                        |
| <b>ACI</b> | <b>n</b> | add immediate plus<br>previous carry |
| <b>SUI</b> | <b>n</b> | subtract immediate                   |
| <b>SBI</b> | <b>n</b> | subtract with borrow<br>immediate    |

The **n** operands of the above instructions denote the eight bit immediate data to be added or subtracted. These operands may be of any form so long as the value can be contained in eight bits. Some of the possible variations are shown below.

|            |            |                             |
|------------|------------|-----------------------------|
| <b>ADI</b> | <b>3</b>   | decimal constant            |
| <b>SUI</b> | <b>@77</b> | octal constant              |
| <b>SBI</b> | <b>:3F</b> | hexadecimal constant        |
| <b>ACI</b> | <b>'H'</b> | ASCII character<br>constant |

The use of immediate addition is shown in example 7-5.

#### Example 7-5

Memory locations **HAROLD** and **HAROLD+1** contain a double precision number. Use the immediate add instructions to add the constant  $57_{10}$  to it.

| <i>label</i> | <i>inst.</i> | <i>operand</i>  |                                        |
|--------------|--------------|-----------------|----------------------------------------|
|              | <b>LDA</b>   | <b>HAROLD</b>   | low order <b>HAROLD</b><br>to A        |
|              | <b>ADI</b>   | <b>57</b>       | add constant <b>57</b> to<br>low order |
|              | <b>STA</b>   | <b>HAROLD</b>   | save new low order                     |
|              | <b>LDA</b>   | <b>HAROLD+1</b> | high order <b>HAROLD</b><br>to A       |
|              | <b>ACI</b>   | <b>0</b>        | propagate carry to<br>high order       |
|              | <b>STA</b>   | <b>HAROLD+1</b> | save new high order                    |
|              | ...          |                 |                                        |

It is most important that the reader understand what went on in example 7-5. If you didn't follow it then you don't understand the propagation of carry and much of the remainder of this book will be a mystery. The idea of the **ACI** with a zero operand is simply to propagate any carry which may have been generated by the **ADI**. The **STA** and **LDA** which follow the **ADI** do *not* alter any of the flags. When **ACI** is executed the state of carry is the same as when the **ADI** instructions finished. Again, do not leave this example until you understand it.

Double precision addition and subtraction can also be done between the HL and another register pair by means of the **DAD**, **DAC** and **DSC** instructions. The operands of these instructions specify the addend register pair, e.g.:

|            |          |                                     |
|------------|----------|-------------------------------------|
| <b>DAD</b> | <b>B</b> | add BC to HL                        |
| <b>DAC</b> | <b>D</b> | Add DE to HL,<br>including previous |



**DSC    B**

carry  
 subtract BC from HL,  
 including previous  
 borrow

**DAD    SP**

add the contents of the  
 16 bit stack pointer  
 register to HL. More  
 will be said of **SP** later.

**DAD** is an 8080 instruction, while **DAC** and **DSC** are unique to the Z80. **DAD** performs addition which ignores previous carry and sets only the carry bit of the flag word, all the other flags being left unchanged. **DAC** and **DSC** include previous carry/borrow and do set the flags in a meaningful way. Note that it is possible to check for arithmetic overflow after **DAC** and **DSC** but not after **DAD**. The use of these double precision instructions is shown in example 7-6.

#### Example 7-6

A pair of quadruple precision numbers is located at **N1** and **N2**. Find their sum and leave it at **Q1**. If overflow occurs, transfer control to location **TILT**.

| <i>label</i> | <i>inst.</i> | <i>operand</i> |                       |
|--------------|--------------|----------------|-----------------------|
|              | <b>LHLD</b>  | <b>N1</b>      | low augend to HL      |
|              | <b>LDED</b>  | <b>N2</b>      | low addend to DE      |
|              | <b>DAD</b>   | <b>D</b>       | form low order sum    |
|              | <b>SHLD</b>  | <b>Q1</b>      | save low sum          |
|              | <b>LHLD</b>  | <b>N1+2</b>    | get high order augend |
|              | <b>LDED</b>  | <b>N2+2</b>    | and high order addend |
|              | <b>DAC</b>   | <b>D</b>       | form high sum         |
|              |              |                | including carry from  |
|              |              |                | low order             |
|              | <b>SHLD</b>  | <b>Q1+2</b>    | save high order sum   |
|              | <b>JV</b>    | <b>TILT</b>    | test for overflow     |
|              | ...          |                |                       |

The two instructions, **DAD** and **DAC**, perform 16 bit addition without and with carry, respectively. For subtraction, however, there is only the **DSC** instruction which always includes previous borrow. To accomplish multilength subtraction, therefore, the carry/borrow must be set to zero before the lowest order subtraction is done. This is shown in example 7-7.

#### Example 7-7

Double precision numbers are located in the HL and DE register pairs. Find the difference DE-HL, leaving it in HL. Notice that the HL contents must be subtracted from the DE contents. This cannot be done directly, so the numbers must be swapped. This is done with the **XCHG** instruction which swaps the contents of DE with those of HL.

| <i>label</i> | <i>inst.</i> | <i>operand</i> |                                                                                                                                                                                               |
|--------------|--------------|----------------|-----------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------|
|              | <b>XCHG</b>  |                | this swaps the contents of the DE and HL register pairs                                                                                                                                       |
|              | <b>RSC</b>   |                | this sets the carry bit to zero. <b>RSC</b> is not an 8080 or Z80 instruction as such, but a synonym for another instruction which has no effect other than to set the flags ( <b>ORA A</b> ) |
|              | <b>DSC</b>   | <b>D</b>       | form the difference                                                                                                                                                                           |
|              | ...          |                |                                                                                                                                                                                               |

Until now we have treated all numbers on the Z80 as if they were integers. An integer is simply a number in which the point is implied to be all the way at the right hand end of the number. This need not be so, as the hardware performs only arithmetic and is indifferent to any arbitrary placement of a point by the programmer. Because

many applications require the representation and manipulation of fractions we will now explore this subject.

The integer portion of a number is represented as the sum of a series of positive powers of two which increases to the left. If we decided to locate the point in the middle of the Z80 word the decimal number 11 would look like this:

1011.0000

A fraction is represented by a series of negative powers of two going toward the right. The leftmost fraction bit corresponds to the  $2^{-1}$  or halves position, the next one to the  $2^{-2}$  or quarters position. In this scheme the decimal number 11.25 would be represented:

1011.0100

The integer is interpreted exactly as before. The fraction consists of a zero in the halves bit, a one in the quarters bit and zeros elsewhere. The total value of the fraction is therefore  $\frac{1}{4}$ . The number of bits reserved for fraction, in the above case 4, is known as the *scale factor*. Numbers with identical scale factors may be added or subtracted just as integers are, e.g.:

$$\begin{array}{rcl} 1011.0100_2 & = & 11.25_{10} \\ + 0001.1100_2 & = & 1.75_{10} \\ \hline 1101.0000_2 & = & 13.0_{10} \end{array}$$

If the numbers in the addition above were to be considered as pure integers, with the point located all the way to the right, their values would be 180 (11.25) and 28 (1.75) and their sum would be 208. If the point in the sum just derived is ignored and the resulting integer converted to decimal, its value is 208. The addition process itself is indifferent to the placement of the point.

If the scale factors of numbers to be added or subtracted are different we have a problem. Before the addition or subtraction can take place the scale factors must be equalized by shifting one or the other of the numbers. Consider the numbers:

000001.01<sub>2</sub>    1.25, scale factor 2  
 00011.100<sub>2</sub>    3.5, scale factor 3

These two numbers cannot be added directly because of the difference in their scale factors. The upper number could be shifted left to give:

00001.010<sub>2</sub>    1.25, scale factor 3

or the lower one could be shifted right to give:

000011.10<sub>2</sub>    3.5, scale factor 2

In this case either scaling would work to allow the numbers to be added or subtracted, but let's go back to the number with which we started the fraction exercise, 1011.0100 and attempt to add 001.10001, like this:

1011.0100    11.25, scale factor 4  
 + 001.10001    1.53125, scale factor 5

Now we have a problem. If we shift the lower number to the right the lowest bit will be lost and the value of the number will become 1.5 instead of 1.53125. If we shift the upper number to the left, however, we are in a worse situation. The highest bit is shifted out and lost, the result being:

011.01000<sub>2</sub>

or 3.25 scale factor 5. Where the loss of the 1/32s bit in the right shift above involved a loss of precision, this left shift is a total disaster which destroys the value of the number.

This type of arithmetic in which the programmer chooses a location for the point and then shifts to equalize scale factors is known as *fixed point arithmetic*. If a problem can be solved using fixed point arithmetic there are significant advantages in both space and speed to be gained. Where this is not possible or where the range of intermediate results cannot be predicted it is necessary to use a system of arithmetic in which the scale factor is kept automatically. This

scheme is known as *floating point arithmetic* and will be covered in depth in a later chapter.

The shifting required for scaling and other purposes in the Z80 is accomplished by three distinct types of shift instructions. These are:

- a) Rotary shifts in which the bit shifted out reenters the register from the opposite end, either directly or through the carry bit;
- b) Logical shifts in which the vacated position is filled with zero and the bit shifted out enters carry;
- c) Arithmetic shifts in which only the rightmost 7 bits participate in the shift, the sign remaining fixed.

The Z80 has two types of rotary shift. In the first type the bit shifted out reenters the other end of the register immediately and is also copied into the carry bit. The basic mnemonics for these shifts are **RRC** and **RLC** for right and left shifts, respectively. If no register name is attached to the mnemonic the A register is shifted. If a register name is attached, e.g., **RRCE**, then the named register is the one shifted. The effect of an **RRC** is shown below.

| carry | register |        |
|-------|----------|--------|
| x     | 01110001 | before |
| 1     | 10111000 |        |

The entire register has been rotated right, the former low bit now becoming the high bit as well as being copied into carry. The former contents of carry are irrelevant and are lost. The effect of an **RLC** is:

| carry | register |        |
|-------|----------|--------|
| x     | 01110001 | before |
| 0     | 11100010 | after  |

The register contents have been rotated left, the former high bit being copied into the low bit and also carry.

Any of the register names A, B, C, D, E, H or L may be appended to the **RRC** and **RLC** mnemonics to perform shifts in those registers. Words in computer memory may be shifted in the same way by spe-



cifying a pointer. The mnemonic form **RRCM** or **RLCM** shifts the memory word pointed to by HL. The forms **RRCX**, **RRCY**, **RLCX** and **RLCY** specify the shifting of a memory word pointed to by X or Y. Note that in this last case the operand must specify a displacement, as for any indexed instruction, i.e.:

**RRCX      0**

The second type of rotary shift uses the carry bit as a ninth participating bit. The bit shifted out of the register enters carry and the former contents of carry enter the opposite end of the word. This is useful in multiple length shifts as we shall presently see. The form of the mnemonics for these shifts is **RrR** and **RrL** for the right and left shifts respectively, the small **r** in the form denoting the name of an eight bit register as specified for **RRC** and **RLC** above. The effect of **RDR** on the D register and carry is shown below:

| <i>carry</i> | <i>D register</i> |        |
|--------------|-------------------|--------|
| 1            | 01110010          | before |
| 0            | 10111001          | after  |

Note that the rotation takes place *through* the carry bit. The former contents of carry are not irrelevant. The effect of an **RHL** on the H register and carry is shown below.

| <i>carry</i> | <i>H register</i> |        |
|--------------|-------------------|--------|
| 0            | 10101010          | before |
| 1            | 01010100          | after  |

Before leaving these nine bit rotary shifts, note must be made of their effect on the flags. **RAR** and **RAL** are 8080 instructions which rotate the A register and carry, with only A and carry being changed. Both are one word instructions. The Z80 contains two word forms of these instructions which perform the same functions as **RAR** and **RAL** but set the sign and zero flags as well as carry. If the two word Z80 form is required, a dollar sign is attached to the mnemonic, i.e., it reads **RAR\$** or **RAL\$**. This applies only to A register rotates.

In summary, the rotary shifts are of two types. The first type, **RRC** and **RLC**, rotates the word as an eight bit unit, the bit shifted

out entering the other end of the word immediately and being copied simultaneously into the carry bit. The former contents of carry are irrelevant and are lost. Eight successive shifts in the same direction restores the original A register, but not carry. In the second type, **RrR** and **RrL**, the carry bit and register are rotated as a nine bit unit, the former contents of carry entering the vacated position and the bit shifted out entering carry. Nine successive shifts in the same direction restores both the register and carry to their original condition.

Logical shifts are similar to rotary shifts except that the vacated position is always filled with zero. The effect of a logical shift can be achieved with a nine bit rotary shift if the carry bit is set to zero before shifting. Mnemonics for the logical shifts are **LLr** and **LRr** for left and right shifts, respectively, the small **r** denoting the name of an eight bit register. If a memory word is to be shifted it must be done through a pointer. To shift the memory word pointed to by HL:

**LLM**  
or  
**LRM**

If the pointer is held in X or Y the mnemonics become **LRMX**, **LRMY**, **LLMX** and **LLMY**, with the displacement being specified in the operand as for all other operations through X and Y, e.g.:

**LLMX 1**

performs a logical left shift of the memory word one higher than that pointed to by X, while:

**LRMY -2**

right shifts the word two below the one pointed to by Y.

The effect of a logical right shift is shown below.

*carry register/memory*

|   |          |        |
|---|----------|--------|
| x | 11111111 | before |
| 1 | 01111111 | after  |

The previous contents of carry are irrelevant and are lost. The effect of a logical left shift is:

| <i>carry register/memory</i> |          |        |
|------------------------------|----------|--------|
| x                            | 11111111 | before |
| 1                            | 11111110 | after  |

The arithmetic effect of logical shifts is to multiply or divide by two, multiplication for left shifts and division for right shifts. For left shifts the multiplication is exact provided no significant bits are shifted out. For right shifts the effect is to divide by two, discarding any remainder. Consider the following right shift.

|                           | <i>carry</i> | <i>register/memory</i> | <i>decimal</i>   |
|---------------------------|--------------|------------------------|------------------|
|                           | x            | 00010101               | 21 <sub>10</sub> |
| after one right shift:    | 1            | 00001010               | 10 <sub>10</sub> |
| after two right shifts:   | 0            | 00000101               | 5 <sub>10</sub>  |
| after three right shifts: | 1            | 00000010               | 2 <sub>10</sub>  |
| after four right shifts:  | 0            | 00000001               | 1 <sub>10</sub>  |
| after five right shifts:  | 1            | 00000000               | 0 <sub>10</sub>  |

In each case the result is the same as if division by two had taken place with the remainder discarded.

The final type of shift is known as an arithmetic shift. In these shifts the sign bit is treated as a special case. In the case of a left shift bits shifted out of the highest magnitude bit do not enter the sign. While the Z80 does not have an arithmetic left shift instruction, the left shift function will be shown here for completeness.

|          |         |                    |
|----------|---------|--------------------|
| 11001001 | before, | - 55 <sub>10</sub> |
| 10010010 | after,  | -110 <sub>10</sub> |

The effect here is the same as that of a logical shift, namely to multiply the number by two. If the arithmetic left shift is performed again we have:

|          |     |
|----------|-----|
| 10100100 | -92 |
|----------|-----|

a wrong result. This is because a significant bit, the zero immediately to the right of the sign, has been lost.

Arithmetic right shift is accomplished in the Z80 by the **ARr** instruction, the small **r** denoting the name of an eight bit register. In an arithmetic right shift only the rightmost seven bits of the number are shifted, with the vacated high magnitude bit being filled with a copy of the sign bit. For positive numbers this is identical to a logical shift. If the number is negative the vacated bit becomes a one, however. An arithmetic right shift of a negative number is shown below.

|                     | carry | register/memory |                                    |
|---------------------|-------|-----------------|------------------------------------|
|                     | x     | 11111011        | -5 <sub>10</sub>                   |
| after one shift:    | 1     | 11111101        | -3 <sub>10</sub>                   |
| after two shifts:   | 1     | 11111110        | -2 <sub>10</sub>                   |
| after three shifts: | 0     | 11111111        | -1 <sub>10</sub>                   |
| after four shifts:  | 1     | 11111111        | -1 <sub>10</sub> , the same thing! |

This produces a result that is rather different from a logical shift. The results of the successive shifts do not simply discard the remainder but *round down to the next more negative integer*. No matter how many arithmetic right shifts follow the result will remain the same, minus one.

A multilength arithmetic right shift is shown in example 7-8.

#### Example 7-8

The double precision signed number at location **POPEYE** is to be divided by 2 by shifting. The Z80's registers are not available so the shift must be done through a pointer held in X.

This problem requires a double right shift. The high order part is shifted first, the bit shifted out being held in carry. The bit in carry is then rotated into the high bit of the low order word using a rotate which includes carry as a ninth bit.

| <i>label</i> | <i>inst.</i> | <i>operand</i>  |                                                                                                                                                                                                                           |
|--------------|--------------|-----------------|---------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------|
|              | <b>LXI</b>   | <b>X,POPEYE</b> | address of the low order word of <b>POPEYE</b> to X                                                                                                                                                                       |
|              | <b>ARMX</b>  | <b>1</b>        | arithmetic shift high order. Recall that the high order occupies the word with memory address one greater than the low order. The displacement of 1 forces this word to be addressed. The bit shifted out is now in carry |
|              | <b>RMRX</b>  | <b>0</b>        | rotate the bit shifted out of the high order into the high bit of the low order word                                                                                                                                      |
|              |              | <b>...</b>      |                                                                                                                                                                                                                           |

A double length left shift involves a similar process. This time the low order is shifted first, the bit shifted out again being held temporarily in carry. This bit is then shifted into the low position of the high order word with a rotary shift. In performing this type of shift it is the programmer's responsibility to insure that no overflow occurs, i.e., that no significant bit is shifted out of the high order. The Z80 cannot detect this automatically and the loss of a high order bit is usually a catastrophe. If the number is an unsigned magnitude the overflow can be detected by testing for carry after the high order is shifted left. For signed numbers a test must be made to see whether the shift has altered the algebraic sign. If the sign has changed the shift is invalid. Some microcomputers like the M6800 detect this condition in hardware by sensing the logical difference (Exclusive OR) of the bits shifted into and out of the sign bit during all left shifts. The signed case is shown in example 7-9.



## Example 7-9

A signed double precision number, i.e., a sign bit and 15 magnitude bits, is in the BC register pair. Perform an arithmetic left shift of this number. If an overflow occurs send control to location **DSASTR**.

The shift itself is simple enough, logical shift the low order word, then “roll” the bit shifted out of the low order word into the low position of the high order word with a nine bit rotate left instruction. The detection of the change in sign is the main problem here. To do this a copy of the original high order is kept and the signs of the unshifted and shifted high orders are compared. If the signs are different an overflow has occurred. The comparison of the two signs is best done by Exclusive ORing them together. This involves use of an instruction not yet discussed, **XRA**. The effect of **XRA** is to Exclusive OR the two quantities, setting the F register flags to show the character of the result. If the two signs were different the result of Exclusive ORing them will be negative. The overflow test can therefore be done with a **JM** instruction. The solution to the problem is:

| <i>label</i> | <i>inst.</i> | <i>operand</i> |                                                                                             |
|--------------|--------------|----------------|---------------------------------------------------------------------------------------------|
|              | <b>MOV</b>   | <b>A,B</b>     | save a copy of the high order in A register                                                 |
|              | <b>LLC</b>   |                | logical shift low order left                                                                |
|              | <b>RBL</b>   |                | shift high order, bring in bit from low order                                               |
|              | <b>XRA</b>   | <b>B</b>       | form Exclusive OR of new and old high orders. If the sign flag is set an overflow occurred. |
|              | <b>JM</b>    | <b>DSASTR</b>  | test for overflow                                                                           |

If the double precision quantity to be left shifted is an unsigned magnitude the detection of overflow is somewhat different. Since a significant bit is in this case a one, the overflow can be detected by testing carry. Carry will contain the bit shifted out. If it is a one an overflow has occurred. This shift is shown in example 7-10.

#### Example 7-10

An unsigned 16 bit number is in the HL registers. Shift this number left and transfer control to **DSASTR** if an overflow occurs.

| <i>label</i> | <i>inst.</i> | <i>operand</i> |                                                |
|--------------|--------------|----------------|------------------------------------------------|
|              | <b>LL</b>    |                | shift low order left                           |
|              | <b>RHL</b>   |                | shift high order, "roll in" bit from low order |
|              | <b>BC</b>    | <b>DSASTR</b>  | test for a one in carry                        |
|              | ...          |                |                                                |

Logical operations can be performed in the Z80 *only* in the A register. The mnemonics for Inclusive OR, AND and Exclusive OR are **ORA**, **ANA** and **XRA** respectively, the trailing **A** in each mnemonic being redundant since no register other than A can be used. The operands for these mnemonics are either the name of an eight bit register or a pointer specification, e.g.:

|            |          |                                   |
|------------|----------|-----------------------------------|
| <b>ORA</b> | <b>L</b> | OR contents of A with L           |
| <b>ANA</b> | <b>B</b> | AND contents of A with B          |
| <b>XRA</b> | <b>C</b> | Exclusive OR contents of A with C |

In each case the result is left in the A register and the other register is undisturbed. To perform logical operations between A and a memory word whose address is in HL:

|            |          |
|------------|----------|
| <b>ORA</b> | <b>M</b> |
| <b>ANA</b> | <b>M</b> |
| <b>XRA</b> | <b>M</b> |

If the pointer is held in X or Y the mnemonic changes, i.e.:

|            |          |          |            |          |
|------------|----------|----------|------------|----------|
| <b>ORX</b> | <b>0</b> | <b>—</b> | <b>ORY</b> | <b>0</b> |
| <b>ANX</b> | <b>0</b> | <b>—</b> | <b>ANY</b> | <b>0</b> |
| <b>XRX</b> | <b>0</b> | <b>—</b> | <b>XRY</b> | <b>0</b> |

An application of the Inclusive OR instruction is shown in example 7-11.

#### Example 7-11

A block of five memory locations beginning at **COUNTS** is being used to accumulate counts of the frequency with which incoming data falls into various ranges. Scan the five word block to determine whether any of the counts has become nonzero. If they are all zero transfer control to location **BACK**. If any or all of the counts are nonzero transfer control to **BINGO**.

| <i>label</i> | <i>inst.</i> | <i>operand</i>  |                                                                                                                                                                                        |
|--------------|--------------|-----------------|----------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------|
|              | <b>LXI</b>   | <b>X,COUNTS</b> | address of first word of block to X                                                                                                                                                    |
|              | <b>ZAR</b>   |                 | clear the A register.<br><b>ZAR</b> is not really an instruction, but a synonym for another instruction, <b>XRA A</b> , which Exclusive ORs A with itself to produce a zero A register |
|              | <b>ORX</b>   | <b>0</b>        | OR first member                                                                                                                                                                        |
|              | <b>ORX</b>   | <b>1</b>        | second . . .                                                                                                                                                                           |
|              | <b>ORX</b>   | <b>2</b>        | third . . .                                                                                                                                                                            |

**ORX** 3  
**ORX** 4

**BZ** **BACK**  
**BRA** **BINGO**

...

fourth . . .  
 and fifth. If all five  
 words were zero the A  
 register is zero and the  
 Z flag will indicate this  
 go to **BACK** if all zeros  
 or to **BINGO** if one or  
 more is nonzero

The **AND** function can be used to scan a block in a different way.  
 This is shown in example 7-12.

#### Example 7-12

The five word block which begins at **CNTGRD** contains the temperatures at various points in a food freezing operation. Scan the block and transfer control to **ALARM** if any of the temperatures is positive, i.e., at or above freezing on the centigrade scale. If all of the temperatures are negative the sign bits of the five words will all be ones. **AND**ing the five words together should produce a result with a one in the sign bit in this case. If one or more of the temperatures is positive or zero this result will have a zero in the sign bit.

| <i>label</i> | <i>inst.</i> | <i>operand</i>  |                                          |
|--------------|--------------|-----------------|------------------------------------------|
|              | <b>LXI</b>   | <b>X,CNTGRD</b> | address of temperature block to X        |
|              | <b>LAX</b>   | <b>0</b>        | first temperature to A through X         |
|              | <b>ANX</b>   | <b>1</b>        | <b>AND</b> the other four with the first |
|              | <b>ANX</b>   | <b>2</b>        | ...                                      |

|            |              |                                                                                      |
|------------|--------------|--------------------------------------------------------------------------------------|
| <b>ANX</b> | <b>3</b>     | ...                                                                                  |
| <b>ANX</b> | <b>4</b>     | ...                                                                                  |
| <b>JP</b>  | <b>ALARM</b> | if any of the<br>temperatures was zero<br>or positive the result<br>will be positive |
| ...        |              |                                                                                      |

The Exclusive OR function is often used for a kind of comparison, e.g., the determination of the algebraic sign of a product or quotient without actually doing the multiplication or division. Another application for this kind of bit comparison is shown in example 7-13.

#### Example 7-13

The trigonometric sine of an angle is in the B register, the cosine of the same angle being in the D register. Determine quadrant in which the angle lies, leaving the result in the H register.

The sine of an angle is positive in the first and second quadrants and negative in the third and fourth, while the cosine is positive in the first and fourth quadrants and negative in the second and third. The algebraic signs of the functions in B and D are used to determine the quadrant.

| <i>label</i> | <i>inst.</i> | <i>operand</i> |                                                                                                                                        |
|--------------|--------------|----------------|----------------------------------------------------------------------------------------------------------------------------------------|
|              | <b>MVI</b>   | <b>H,0</b>     | clear H to receive result                                                                                                              |
|              | <b>MOV</b>   | <b>A,B</b>     | bring sine into A                                                                                                                      |
|              | <b>XRA</b>   | <b>D</b>       | this produces the logical<br>difference of the alge-<br>braic signs of the sine<br>and cosine in the sign<br>bit of A. If this is zero |



**RAL**

**RHL**

**MOV A,B**

**ORA A**

**JP QUAD2**

**INR H**

**QUAD2 INR H**  
**INR H**

...

the angle must lie in the first or third quadrant. If the sign bit of A is a one the angle falls in the second or fourth quadrant

now rotate the sign bit out of A

and into the low bit of H. The H register now contains zero or one

get the sine again this has no effect on A.

It merely sets the flags so that the algebraic sign can be tested

if the sine was positive there is nothing more to be done except to increment the quadrant number to read one or two instead of zero or one

if the sine was negative the quadrant number must be incremented twice as the angle lies in the third or fourth quadrant

this final **INR** increments the quadrant number to the one through four range

Like the load and arithmetic instructions, the logical instructions have immediate forms in which the operand is part of the instruction. These forms are **ORI**, **ANI** and **XRI** for OR, AND and Exclusive OR, respectively. The immediate operand may be of any form which generates a legitimate eight bit number, e.g.:

|            |                     |                                                                                                                    |
|------------|---------------------|--------------------------------------------------------------------------------------------------------------------|
| <b>ORI</b> | <b>'0'</b>          | ASCII operand                                                                                                      |
| <b>ANI</b> | <b>:F</b>           | hexadecimal operand                                                                                                |
| <b>XRI</b> | <b>@66</b>          | octal operand                                                                                                      |
| <b>ANI</b> | <b>ORPHAN</b>       | symbolic operand.<br><b>ORPHAN</b> would, in this case, be defined by an <b>EQU</b> like<br><b>ORPHAN EQU :CO.</b> |
| <b>XRI</b> | <b>FIRST-SECOND</b> | sum or difference of symbols                                                                                       |

In all of these cases the operand must fit into eight bits. If the operand length is greater than eight bits the instruction will be flagged by the assembler with an L, for length error.

The final topics of this chapter are the comparison and negation operations. Comparison is performed on the Z80 by the **CMP** and **CPI** instructions. Like the logical instructions, comparison can be done only between the A register and some other entity, either another eight bit register or a memory word indicated by a pointer. Comparison of A to another register is done by naming the other register as operand of the **CMP**, e.g.:

**CMP B**  
or  
**CMP L**

compare A to the B and L registers. Comparison of A to a memory word pointed to by HL is written:

**CMP M**

while comparison through the X or Y registers is done by **CMPX** and

**CMPY** with a displacement specified as the operand.

A comparison is performed by subtracting the contents of the word being compared from the contents of the A register. The difference between a compare and a subtract is that the contents of A are not changed. The result of the subtraction never appears, but is used to set the flags of the F register. These flags can then be tested just as if a subtraction had actually taken place. Thus if A contained a 1 and C contained a 2, the instruction:

**CMP    C**

would set the flags as if the result -1 had been generated in A. Both carry and the sign flag would be set to -1, and the Z flag set to zero.

The immediate form of compare, **CPI**, can be used to compare the A register to some constant value, e.g.:

**CPI    7**

compares A to the constant 7, and:

**CPI    'T'**

compares A to the ASCII code for the letter T, the flags being set to reflect the outcome of the comparison. A simple comparison is shown in example 7-14.

#### Example 7-14

A memory location named **TMPCHR** contains a Celsius temperature of a fluid in a chemical process. Compare the contents of **TMPCHR** to the contents of the memory word named **VALUE**. If **TMPCHR** is greater than or equal to **VALUE** transfer control to location **TOOHOT**, otherwise transfer control to location **TUCOLD**.

| <i>label</i> | <i>inst.</i> | <i>operand</i> |                    |
|--------------|--------------|----------------|--------------------|
|              | <b>LXI</b>   | <b>H,VALUE</b> | address of word to |

**LDA      TMPCHR**

**CMP      M**

**JP        TOOHOT**

**BRA      TUCOLD**

**...**

which **TMPCHR** is to be compared to **HL** Celsius temperature to **A** perform the comparison. The flags are now set as if **VALUE** had been subtracted from **TMPCHR**. If **TMPCHR** would have been greater than or equal to **VALUE** the result would have been positive test for positive otherwise go to **TUCOLD**

The conditions of example 7-14 presumed the quantities involved to be signed numbers. The testing jump instruction is fairly easy to conceptualize in this case. If the quantities being compared were eight bit magnitudes the test condition would have been different. This type of comparison is shown in example 7-15.

#### Example 7-15

The ASCII codes for the alphabet A-Z are represented by the 26 codes **C1<sub>16</sub>** to **DA<sub>16</sub>**. A single ASCII character is in location **LETTER**. Test the character in **LETTER**. If it is an M transfer control to location **EM**. If it falls earlier in the alphabet than M go to **BEFORE** and if it falls after M go to **AFTER**.

| <i>label</i> | <i>inst.</i> | <i>operand</i> |                                                                                                   |
|--------------|--------------|----------------|---------------------------------------------------------------------------------------------------|
|              | <b>LDA</b>   | <b>LETTER</b>  | character to be tested to A                                                                       |
|              | <b>CPI</b>   | <b>'M'</b>     | compare to ASCII M                                                                                |
|              | <b>BZ</b>    | <b>EM</b>      | if the Z flag is set the compare was identical, i.e., <b>LETTER</b> contains the ASCII code for M |
|              | <b>BNC</b>   | <b>AFTER</b>   | if the "subtraction" did not set the carry bit the character came after M                         |
|              | <b>BRA</b>   | <b>BEFORE</b>  | otherwise it must have come before                                                                |
|              | ...          |                |                                                                                                   |

This testing of the condition bits after comparison is a tedious business, involving as it does remembering which bit represents which condition. This involves an unnecessary burden on memory and the assembler can help. Consider the sequence in example 7-15 above, but rewritten this way:

|            |               |                                 |
|------------|---------------|---------------------------------|
| <b>LDA</b> | <b>LETTER</b> |                                 |
| <b>CPI</b> | <b>'M'</b>    |                                 |
| <b>BEQ</b> | <b>EM</b>     | branch if equal                 |
| <b>BGE</b> | <b>AFTER</b>  | branch if greater than or equal |
| <b>BRA</b> | <b>BEFORE</b> |                                 |

The assembler will process the mnemonics **BEQ** and **BGE** as synonyms for **BZ** and **BNC**. The assembler will in fact process synonyms for all of the relevant jump and branch combinations after comparisons of either signed numbers or unsigned magnitudes. These synonyms are:

For unsigned eight bit magnitudes:



| <i>mnemonic</i>   | <i>meaning</i>                         | <i>assembles as</i> |
|-------------------|----------------------------------------|---------------------|
| <b>JEQ or BEQ</b> | jump/branch if equal                   | <b>JZ or BZ</b>     |
| <b>JNE or BNE</b> | jump/branch if not equal               | <b>JNZ or BNZ</b>   |
| <b>JLS or BLS</b> | jump/branch if A is less               | <b>JC or BC</b>     |
| <b>JGE or BGE</b> | jump/branch if A greater than or equal | <b>JNC or BNC</b>   |

For signed two's complement numbers:

|                   |                                 |                   |
|-------------------|---------------------------------|-------------------|
| <b>JEQ or BEQ</b> | jump/branch if equal            | <b>JZ or BZ</b>   |
| <b>JNE or BNE</b> | jump/branch if not equal        | <b>JNZ or BNZ</b> |
| <b>J &lt;</b>     | jump if A is less               | <b>JM</b>         |
| <b>J &gt; =</b>   | jump if A greater than or equal | <b>JP</b>         |

Using these synonyms the programmer is relieved of the nuisance of remembering which bit does what after compares. The equal and unequal tests **JEQ/BEQ** and **JNE/BNE** are the same for both signed and unsigned numbers. Note that the Z80 provides no branch equivalents for the **J <** and **J > =** instructions. The longer jump form must be used in both of these cases.

A very common requirement in programs of all types is comparison between limits. One such comparison is shown in example 7-16.

#### Example 7-16

The ASCII representations for the digits 0 through 9 are codes  $B0_{16}$  through  $B9_{16}$ . A character is in the A register. Compare it to these limits, sending control to **OK** if it falls in the 0-9 range and to **NG** if it does not.

| <i>label</i> | <i>inst.</i> | <i>operand</i> |                                                                                               |
|--------------|--------------|----------------|-----------------------------------------------------------------------------------------------|
|              | <b>CPI</b>   | <b>:B0</b>     | compare to lower limit<br>if the character is less<br>than $B0_{16}$ it is<br>eliminated here |
|              | <b>BLS</b>   | <b>NG</b>      |                                                                                               |
|              | <b>CPI</b>   | <b>:BA</b>     |                                                                                               |
|              |              |                | this comparison is to a                                                                       |

**BGE NG**

**BRA OK**

...

value one greater than the code for 9. This is because it is not possible to jump or branch on the less than or equal to condition, only on less than or on greater than or equal to if control gets past here the character was in the 0-9 range

Comparisons of double precision numbers are a bit more complicated. The high order parts are compared first and, only if the high orders are identical, are the low orders examined. An unsigned compare of this type is shown in example 7-17.

#### Example 7-17

Unsigned double precision numbers are in the BC register pair and in memory location **CPND**. The numbers are to be compared with control going to **GRTR** if the BC held number is greater, to **LESS** if the number at **CPND** is greater and to **EQUAL** if the numbers are identical.

| <i>label</i> | <i>inst.</i> | <i>operand</i> |                          |
|--------------|--------------|----------------|--------------------------|
|              | <b>LXI</b>   | <b>X,CPND</b>  | <b>CPND</b> pointer to X |
|              | <b>MOV</b>   | <b>A,B</b>     | get high order           |
|              | <b>CMPX</b>  | <b>1</b>       | compare high orders. If  |
|              |              |                | and only if the high     |
|              |              |                | orders are equal are the |
|              |              |                | low orders examined      |

|               |               |               |                                                                                                                                                     |
|---------------|---------------|---------------|-----------------------------------------------------------------------------------------------------------------------------------------------------|
|               | <b>BEQ</b>    | <b>TSTLOW</b> | skip to low test if equal                                                                                                                           |
|               | <b>BGE</b>    | <b>GRTR</b>   | the equal case is already<br>eliminated by the <b>BEQ</b><br>above, so this <b>BGE</b><br>transfers control only if<br>the BC number was<br>greater |
|               | <b>BRA</b>    | <b>LESS</b>   | all other possibilities<br>eliminated                                                                                                               |
| <b>TSTLOW</b> | <b>MOV</b>    | <b>A,C</b>    | high orders equal, test<br>low orders                                                                                                               |
|               | <b>CMPX 0</b> |               | compare low orders                                                                                                                                  |
|               | <b>BEQ</b>    | <b>EQUAL</b>  | if the low orders are<br>also equal the numbers<br>are identical                                                                                    |
|               | <b>BGE</b>    | <b>GRTR</b>   | same comparison as for<br>high order                                                                                                                |
|               | <b>BRA</b>    | <b>LESS</b>   | BC number less than                                                                                                                                 |
|               |               |               | <b>CPND</b>                                                                                                                                         |
|               |               |               | ...                                                                                                                                                 |

The case of the signed double precision number is only slightly more complicated. The high orders, consisting of sign and seven magnitude bits, must be compared and the tests for signed numbers applied. If the high orders are identical the low orders are tested as before. The low orders of these numbers are pure magnitude, however, and the unsigned tests must be applied. A comparison of this type is shown in example 7-18.

#### Example 7-18

Double precision signed numbers are in memory at locations **BALL** and **STRIKE**. If **BALL** is greater than **STRIKE** send control to **WALK**. If they are equal go to **WAIT** and if **STRIKE** is greater than **BALL** go to **OUT**.

| <i>label</i>  | <i>inst.</i> | <i>operand</i>  |                              |
|---------------|--------------|-----------------|------------------------------|
|               | <b>LXI</b>   | <b>X,BALL</b>   | <b>BALL</b> pointer to X     |
|               | <b>LXI</b>   | <b>Y,STRIKE</b> | <b>STRIKE</b> pointer to Y   |
|               | <b>LAX</b>   | <b>1</b>        | fetch high order <b>BALL</b> |
|               | <b>CMPLY</b> | <b>1</b>        | compare high order           |
|               |              |                 | <b>STRIKE</b>                |
|               | <b>BEQ</b>   | <b>TRYLOW</b>   | high orders equal, try       |
|               |              |                 | low orders                   |
|               | <b>J&lt;</b> | <b>OUT</b>      | <b>BALL</b> less than        |
|               |              |                 | <b>STRIKE</b>                |
|               | <b>BRA</b>   | <b>WALK</b>     | <b>BALL</b> greater than     |
|               |              |                 | <b>STRIKE</b>                |
| <b>TRYLOW</b> | <b>LAX</b>   | <b>0</b>        | fetch low order <b>BALL</b>  |
|               | <b>CMPLY</b> | <b>0</b>        | compare low orders.          |
|               |              |                 | From this point on the       |
|               |              |                 | testing proceeds as          |
|               |              |                 | before                       |
|               | <b>BEQ</b>   | <b>WAIT</b>     | numbers identical            |
|               | <b>BGE</b>   | <b>WALK</b>     | <b>BALL</b> greater than     |
|               |              |                 | <b>STRIKE</b>                |
|               | <b>BRA</b>   | <b>OUT</b>      | <b>BALL</b> less than        |
|               |              |                 | <b>STRIKE</b>                |
|               |              |                 | ...                          |

Negation is performed on the Z80 in single precision by means of the **NEG** instruction. **NEG** forms the two's complement of the contents of the A register. The **NEG** instruction requires two words and is equivalent to the 8080 sequence:

|            |          |                        |
|------------|----------|------------------------|
| <b>CMA</b> |          | one's complement the A |
|            |          | register               |
| <b>INR</b> | <b>A</b> | increment A register   |

The assembler will recognize the pseudo-op **TCA** (two's complement A) and supply the **CMA** and **INR A** sequence. Either the Z80 **NEG** or the 8080 **TCA** can be used to accomplish single precision negation

in A. If 8080 compatibility is to be preserved use **TCA** or the **CMA - INR A** sequence.

Negation of a multiple precision number is a little more complicated. The various orders of the number are one's complemented and the low order then incremented. If this incrementation causes a carry out of the low order the next higher order must also be incremented and so on until there are no more carries to be propagated or the highest order is reached. It is *very* important to check for overflow after any negation. If the number being negated was the maximum negative number in any precision the result will be the maximum negative number with overflow turned on. You neglect to make this overflow test at your immediate peril if there is any possibility that the number being negated can achieve this value. A double precision negation is shown in example 7-19.

#### Example 7-19

A program is required which will negate the 16 bit number in HL. If an overflow occurs control should go to location **OFLO**. Disturb nothing but the A register.

| <i>label</i>  | <i>inst.</i> | <i>operand</i> |                      |
|---------------|--------------|----------------|----------------------|
| <b>DBLNEG</b> | <b>MOV</b>   | <b>A,L</b>     | low order to A       |
|               | <b>CMA</b>   |                | one's complement     |
|               | <b>MOV</b>   | <b>L,A</b>     | return to L          |
|               | <b>MOV</b>   | <b>A,H</b>     | high order to A      |
|               | <b>CMA</b>   |                | one's complement     |
|               | <b>MOV</b>   | <b>H,A</b>     | return to H          |
|               | <b>INR</b>   | <b>L</b>       | increment low order. |

Here again we have no way to test directly for a carry out of the low order. Since the only circumstance in which a carry could occur is if the low order were  $1111111_2$  before the



**BNZ OK**  
**INR H**  
**JV OFLO**  
**OK ...**

**INR L**, the equivalent  
 test is a test for zero  
 after the **INR**  
 no high order increment  
 if low order not zero  
 increment high order  
 test for overflow  
 program resumes

## 8 / SOFTWARE MULTIPLICATION AND DIVISION

Multiplication is vexation,  
Division is as bad;  
The rule of three doth puzzle me,  
And practice drives me mad.

Anonymous Elizabethan Manuscript, 1570

The Z80, like most microcomputers, has no hardware facilities for multiplication and division. The job must therefore be done by programming and it is the purpose of this chapter to show the reader how this is done in general, and some shortcuts for special cases. We can begin by showing one of the oldest tricks in programming. Take any number, say 6, and write it in binary:

00000110

now shift the number left one bit:

00001100

and save this result somewhere. Shift the number left again:

00011000

and once again:

00110000

now add the number we saved above to this last result:

$$\begin{array}{r}
 00110000 \\
 + 00001100 \\
 \hline
 00111100 = 60_{10}
 \end{array}$$

This strange looking procedure has multiplied the original 6 by 10 to give 60. This scheme will work for any number whatever, so long as the capacity of the word is not exceeded. Understanding just why it works is at the root of understanding multiplication in general.

Shifting left once multiplies the number by two. After saving this result, twice the original number, we shifted left twice more, for a total of three left shifts of the original number. Shifting left three times multiplies by  $2^3$  or eight. Adding this to the saved result amounts to adding twice the original number to eight times the original number. Since  $8 + 2 = 10$ , we have multiplied by ten.

To understand how this works, look at the binary representation of the number ten:

00001010

If this number is scanned from right to left, the ones occur in positions corresponding to those which entered into the sum during the shifting. This scheme can be broadened to multiply by any constant whatever. To multiply by twelve, for example consider the binary constant twelve, i.e.:

00001100

There are ones in the  $2^2$  and  $2^3$  positions. The multiplication is accomplished by shifting the number left twice to put its original rightmost bit into the  $2^2$  position, saving this result, and then shifting left again until the original rightmost bit is in the  $2^3$  position. Adding this last number to the saved previous result yields twelve times the original number. Suppose we start with 7:

00000111

and shift left once:

00001110

and left again:

00011100

and save this result. Now shift left once more:

00111000

and add what was just saved:

$$\begin{array}{r} 00111000 \\ + 00011100 \\ \hline 01010100 \end{array} = 84_{10}$$

We have multiplied by twelve simply by shifting and adding.

A multiplication by a constant using a scheme like this is shown in example 8-1.

#### Example 8-1

A number is in the Z80's A register. Multiply it by  $15_{10}$ .

There are a number of ways to go about this. The simplest is to follow the strategy outlined above. Writing 15 in binary:

00001111

shows it to be simply the sum of the first four powers of two. The program therefore looks like this:

| <i>label</i> | <i>inst.</i> | <i>operand</i> |                  |
|--------------|--------------|----------------|------------------|
|              | <b>MOV</b>   | <b>B,A</b>     | save $2^0$ value |
|              | <b>LLA</b>   |                | shift left       |
|              | <b>MOV</b>   | <b>C,A</b>     | save $2^1$ value |
|              | <b>LLA</b>   |                | left again       |
|              | <b>MOV</b>   | <b>D,A</b>     | save $2^2$ value |

|            |          |                                             |
|------------|----------|---------------------------------------------|
| <b>LLA</b> |          | left again to generate $2^3$                |
|            |          | times the original number. The intermediate |
|            |          | results saved in B, C,                      |
|            |          | and D are now added                         |
| <b>ADD</b> | <b>B</b> | ...                                         |
| <b>ADD</b> | <b>C</b> | ...                                         |
| <b>ADD</b> | <b>D</b> | ...                                         |
| ...        |          | program continues                           |

This is the most straightforward way to do the job, but there is yet another approach which is shorter. Multiplication by 16 is easy, involving only a left shift of four places. Since  $15 = 16 - 1$ , shifting left four places and then subtracting the original number would do the job. This is shown below.

|            |            |                         |
|------------|------------|-------------------------|
| <b>MOV</b> | <b>B,A</b> | save original number    |
| <b>LLA</b> |            | now multiply by 16 by   |
| <b>LLA</b> |            | shifting the original   |
| <b>LLA</b> |            | number left four places |
| <b>LLA</b> |            | ...                     |
| <b>SUB</b> | <b>B</b>   | and subtract the        |
|            |            | original value          |
| ...        |            |                         |

Some such scheme can be found to multiply by any constant and in programs which have time constraints they are well worth looking for, since this method is much faster than a full software multiply.

The general case of multiplication is simply a generalization of the process just shown. The numbers to be multiplied are called the *multiplier* and *multiplicand*, the result being known as the *product*. If multiplier and multiplicand are both eight bits in length the product will require 16 bits. In general an N bit multiplier and an M bit multiplicand generate an N+M bit product. The successive bits of the multiplier are tested and the multiplicand added to the product if the



tested bit is a one. If the tested bit is a zero the addition is skipped. The product is then shifted to be in position to receive a contribution from the next power of two if that bit is a one in the multiplier. This process can be carried out from left to right or right to left. Since the double precision product must be shifted in the same direction as the multiplier the optimum way for the Z80 is from left to right, a double right shift being more complex than a double left shift. Double left shift of HL on the Z80 can be done with a **DAD H** instruction, while double right shift requires two instructions. A general 8 bit by 8 bit multiply is shown in example 8-2.

### Example 8-2

Two eight bit unsigned numbers are in the D and E registers. Find their product and leave it in HL.

| <i>label</i> | <i>inst.</i> | <i>operand</i> |                          |
|--------------|--------------|----------------|--------------------------|
|              | <b>MOV</b>   | <b>A,D</b>     | multiplier to A register |
|              | <b>MVI</b>   | <b>D,0</b>     | clear D. This is         |
|              |              |                | necessary because the    |
|              |              |                | DE pair will be used to  |
|              |              |                | perform the double pre-  |
|              |              |                | cision addition of the   |
|              |              |                | multiplicand to the      |
|              |              |                | product being built in   |
|              |              |                | the HL register pair     |
|              | <b>MVI</b>   | <b>B,8</b>     | count to B register      |
|              | <b>LXI</b>   | <b>H,0</b>     | clear product area       |
| <b>M1</b>    | <b>DAD</b>   | <b>H</b>       | shift product left       |
|              | <b>RAL</b>   |                | multiplier bit to carry  |
|              | <b>BNC</b>   | <b>M2</b>      | skip addition if         |
|              |              |                | multiplier bit zero      |
|              | <b>DAD</b>   | <b>D</b>       | otherwise add            |
|              |              |                | multiplicand to product  |
| <b>M2</b>    | <b>DBNZ</b>  | <b>M1</b>      | decrement count and      |
|              |              |                | repeat if not yet zero.  |

Multiplication of signed numbers may be done in a number of ways. The simplest of these is to force both multiplier and multiplicand positive, perform the multiplication as shown in example 8-2, then make the necessary adjustment to the final product. The multiplication of signed numbers is shown in example 8-3.

### Example 8-3

A pair of signed two's complement numbers is in the D and E registers. Find their properly signed product, leaving this result in memory locations **PROD** and **PROD + 1**.

| <i>label</i> | <i>inst.</i> | <i>operand</i> |                               |
|--------------|--------------|----------------|-------------------------------|
| MPY          | MOV          | A,D            | get multiplier                |
|              | XRA          | E              | form product sign             |
|              | STA          | SIGN           | and save it                   |
|              | MOV          | A,D            | now force multiplier positive |
| R2           | ORA          | A              | ...                           |
|              | JP           | R2             | ...                           |
|              | NEG          |                | ...                           |
|              | MOV          | D,A            | and replace it                |
|              | MOV          | A,E            | get multiplicand              |
|              | ORA          | A              | and force it positive         |
|              | JP           | R3             | ...                           |
| R3           | NEG          |                | ...                           |
|              | MOV          | E,A            | and replace it                |
|              | MOV          | A,D            | move multiplier to A          |
|              | MVI          | D,0            | same process as example 8-2   |
| R4           | MVI          | B,8            | ...                           |
|              | LXI          | H,0            |                               |
|              | DAD          | H              |                               |
|              | RAL          |                |                               |
|              | BNC          | R5             |                               |
|              | DAD          | D              |                               |

|              |             |              |                            |
|--------------|-------------|--------------|----------------------------|
| <b>R5</b>    | <b>DBNZ</b> | <b>R4</b>    |                            |
|              | <b>LDA</b>  | <b>SIGN</b>  | check for negative product |
|              | <b>ORA</b>  | <b>A</b>     |                            |
|              | <b>JP</b>   | <b>EOMPY</b> | skip if positive           |
|              | <b>MOV</b>  | <b>A,L</b>   | otherwise negate product   |
|              | <b>CMA</b>  |              |                            |
|              | <b>MOV</b>  | <b>L,A</b>   | ...                        |
|              | <b>MOV</b>  | <b>A,H</b>   |                            |
|              | <b>CMA</b>  |              |                            |
|              | <b>MOV</b>  | <b>H,A</b>   |                            |
|              | <b>INX</b>  | <b>H</b>     |                            |
| <b>EOMPY</b> | <b>SHLD</b> | <b>PROD</b>  | save final product         |
| ...          |             |              | program continues          |

It is worthwhile at this point to take a slight tangent from the subject of multiplication to explore a matter of good programming practice. A close look at example 8-3 will show a couple of places in which more instructions were used than actually needed. The basic multiply loop of example 8-2 was incorporated directly into the signed multiplication of example 8-3. First notice location **R3**, the **MOV A,D** instruction. This follows immediately after both numbers are forced positive, D first and then E. If the work had been arranged in the opposite order we could have arranged for the multiplier to be in A when **R2** was reached, eliminating the necessity for the **MOV A,D** instruction. Also look at the negation of the double precision result which begins with **MOV A,L**. This double negation is a simplification of the procedure used in example 7-19, the check for overflow having been eliminated since no overflow is possible here. That example contained the restriction that only the A register could be used for the negation. Since there is no such restriction here a shorter method for the double negation can be used. The same function rewritten for greater efficiency is shown in example 8-4.

## Example 8-4

| <i>label</i> | <i>inst.</i> | <i>operand</i> |                                |
|--------------|--------------|----------------|--------------------------------|
| <b>MPY</b>   | <b>MOV</b>   | <b>A,D</b>     | form product sign<br>as before |
|              | <b>XRA</b>   | <b>E</b>       | ...                            |
|              | <b>STA</b>   | <b>SIGN</b>    | ...                            |
|              | <b>MOV</b>   | <b>A,E</b>     | then force both                |
|              | <b>ORA</b>   | <b>A</b>       | numbers positive but           |
|              |              |                | this time do E first           |
|              | <b>JP</b>    | <b>R2</b>      | ...                            |
|              | <b>NEG</b>   |                | ...                            |
|              | <b>MOV</b>   | <b>E,A</b>     | ...                            |
| <b>R2</b>    | <b>MOV</b>   | <b>A,D</b>     | then D, so that when           |
|              | <b>ORA</b>   | <b>A</b>       | this is finished, we have      |
|              | <b>JP</b>    | <b>R3</b>      | saved two instructions,        |
|              | <b>NEG</b>   |                | the <b>MOV D,A</b> which       |
| <b>R3</b>    | <b>MVI</b>   | <b>D,0</b>     | replaced the positive          |
|              | <b>MVI</b>   | <b>B,8</b>     | multiplier in D and the        |
|              | <b>LXI</b>   | <b>H,0</b>     | <b>MOV A,D</b> which           |
|              |              |                | brought it back to A           |
| <b>R4</b>    | <b>DAD</b>   | <b>H</b>       | the multiplication loop        |
|              | <b>RAL</b>   |                | remains the same.              |
|              | <b>BNC</b>   | <b>R5</b>      | ...                            |
|              | <b>DAD</b>   | <b>D</b>       | ...                            |
| <b>R5</b>    | <b>DBNZ</b>  | <b>R4</b>      |                                |
|              | <b>LDA</b>   | <b>SIGN</b>    | but the adjustment of          |
|              | <b>ORA</b>   | <b>A</b>       | the product sign can be        |
|              | <b>JP</b>    | <b>EOMPY</b>   | made shorter, since we         |
|              |              |                | can use DE                     |
|              | <b>MVI</b>   | <b>E,0</b>     | clear E. D is already          |
|              |              |                | zero                           |
|              | <b>XCHG</b>  |                | swap DE and HL. HL             |
|              |              |                | is now zero                    |
|              | <b>RSC</b>   |                | clear carry, and               |
|              | <b>DSC</b>   | <b>D</b>       | subtract the product           |
|              |              |                | from zero                      |
| <b>EOMPY</b> | <b>SHLD</b>  | <b>PROD</b>    | store as before                |

Even more could have been saved in example 8-4 by employing a technique which will be described in chapter 9. The **STA SIGN** and both the **LDA SIGN** and the **ORA A** which followed it which saved and retrieved the sign indicator could have been replaced by two one word instructions which save A and the flags and restore them when called for.

Double precision multiplication can be accomplished in a number of ways. The simplest is to use the double addition and register rotates of the Z80 to perform the exact double precision parallel of the procedure used in example 8-4. Multiplier and multiplicand are each 16 bits long in this case, so room must be left for a 32 bit product. The procedure is shown in example 8-5. A new instruction is used in this example, **X+X**. This instruction adds the contents of X to itself, effectively doubling the value in X. This is exactly equivalent to shifting X left one bit. If the addition of X to itself causes a carry out of the high bit of X, it is held for testing in the carry bit.

#### Example 8-5

16 bit unsigned magnitudes are held in the X register and in the DE register pair. Find their product and leave it in the BC and HL register pairs, high order in BC.

| <i>label</i> | <i>inst.</i> | <i>operand</i> |                                   |
|--------------|--------------|----------------|-----------------------------------|
| <b>SM</b>    | <b>LXI</b>   | <b>H,0</b>     | clear low product                 |
|              | <b>MOV</b>   | <b>B,H</b>     | clear high product                |
|              | <b>MOV</b>   | <b>C,H</b>     | multiply step count to A register |
| <b>DM1</b>   | <b>MVI</b>   | <b>A,16</b>    |                                   |
|              | <b>DAD</b>   | <b>H</b>       | shift low product left            |
|              | <b>RCL</b>   |                | propagate to high product         |
|              | <b>RBL</b>   |                | ...                               |
|              | <b>X+X</b>   |                | shift multiplier left             |
|              | <b>BNC</b>   | <b>DM2</b>     | skip if multiplier bit zero       |
|              | <b>DAD</b>   | <b>D</b>       | otherwise add                     |



**BNC DM2**

**INR C**

multiplicand  
test for carry to higher  
order(s)  
propagate carry to third  
order. Now we have a  
problem. **INR** does not  
affect the carry bit, but  
some kind of test must  
be made to see if there  
was a carry out of C  
which must be propa-  
gated into the highest  
order in B. The only sit-  
uation in which incre-  
menting C could cause  
a carry out of its high  
bit is if C contained  
 $1111111_2$  before the in-  
crementation. The re-  
sult of the incrementa-  
tion would be  
 $0000000_2$ . The carry  
condition would be met  
in this case, and in this  
case only, by a test  
for zero after the  
incrementation  
skip if no carry from C  
propagate carry to B  
decrement multiply step  
count. Since the B regis-  
ter is occupied by the  
high order product we  
can't use it to hold the  
count and take ad-  
vantage of the **DBNZ**  
instruction

**BNZ DM2**

**INR B**

**DM2 DCR A**

|            |            |                       |
|------------|------------|-----------------------|
| <b>BNZ</b> | <b>DM1</b> | another step if count |
|            |            | not zero              |
| ...        |            |                       |

Example 8-5 was an exact double precision parallel of the process shown in single precision earlier. A refinement can be made to it, however, which will make it even more efficient. This depends upon the fact that the high product is initially zero, the carries from the low product being shifted into it as they are generated in each multiply step. If the multiplier were to be held in the high product area the initial product shift could be made to serve a dual purpose, to shift the product and to shift a new multiplier bit into carry to be tested. This would eliminate the use of X entirely. The multiplication rewritten to take advantage of this is shown in example 8-6.

#### Example 8-6

Double precision unsigned numbers are in the BC and DE register pairs. Find their product, leaving the high order in BC and the low order in HL.

| <i>label</i> | <i>inst.</i> | <i>operand</i> |                          |
|--------------|--------------|----------------|--------------------------|
| <b>DMPY</b>  | <b>MVI</b>   | <b>A,16</b>    | count to A               |
|              | <b>LXI</b>   | <b>H,0</b>     | clear low product        |
| <b>DMPY1</b> | <b>DAD</b>   | <b>H</b>       | shift product left       |
|              | <b>RCL</b>   |                | shift low product bit    |
|              | <b>RBL</b>   |                | into high product,       |
|              |              |                | and multiplier bit into  |
|              |              |                | carry                    |
|              | <b>BNC</b>   | <b>DMPY2</b>   | skip if multiplier bit   |
|              |              |                | zero                     |
|              | <b>DAD</b>   | <b>D</b>       | otherwise add            |
|              |              |                | multiplicand             |
|              | <b>BNC</b>   | <b>DMPY2</b>   | skip if no carry to high |
|              |              |                | orders                   |

|              |            |              |                        |
|--------------|------------|--------------|------------------------|
|              | <b>INR</b> | <b>C</b>     | increment third order  |
|              | <b>BNZ</b> | <b>DMPY2</b> | skip if no carry to    |
|              |            |              | fourth order           |
|              | <b>INR</b> | <b>B</b>     | increment fourth order |
| <b>DMPY2</b> | <b>DCR</b> | <b>A</b>     | decrement count        |
|              | <b>BNZ</b> | <b>DMPY1</b> | and do it again if not |
|              |            |              | zero                   |
|              | <b>...</b> |              |                        |

The program in example 8-6 is both shorter and faster than the one in example 8-5. It would be an excellent test of the reader's understanding of this material to go back and modify examples 8-3 and 8-4 to take advantage of the same economies which were made in example 8-6. Double precision signed multiplication is done by the same general procedure as for single precision in example 8-4. The only new twist is that the negation of the product must be done in quadruple precision. The process is shown in example 8-7.

#### Example 8-7

Signed double precision numbers are in the BC and DE register pairs. Find their product, leaving the high order in BC and the low order in HL.

|              |              |                |                          |
|--------------|--------------|----------------|--------------------------|
| <i>label</i> | <i>inst.</i> | <i>operand</i> |                          |
| <b>SDM</b>   | <b>MOV</b>   | <b>A,B</b>     | determine product sign   |
|              | <b>XRA</b>   | <b>D</b>       | ...                      |
|              | <b>STA</b>   | <b>SIGN</b>    | and save it              |
|              | <b>MOV</b>   | <b>A,B</b>     | get high order           |
|              |              |                | multiplier               |
|              | <b>ORA</b>   | <b>A</b>       | test algebraic sign      |
|              | <b>JP</b>    | <b>SDM1</b>    | skip if already positive |
|              | <b>ZAR</b>   |                | clear A for subtract.    |
|              |              |                | The multiplier will be   |

|             |             |             |
|-------------|-------------|-------------|
|             | <b>SUB</b>  | <b>C</b>    |
|             | <b>MOV</b>  | <b>C,A</b>  |
|             | <b>MVI</b>  | <b>A,0</b>  |
|             | <b>SBB</b>  | <b>B</b>    |
|             | <b>MOV</b>  | <b>B,A</b>  |
| <b>SDM1</b> | <b>MOV</b>  | <b>A,D</b>  |
|             | <b>ORA</b>  | <b>A</b>    |
|             | <b>JP</b>   | <b>SDM2</b> |
|             | <b>LXI</b>  | <b>H,0</b>  |
|             | <b>DSC</b>  | <b>D</b>    |
|             | <b>XCHG</b> |             |
| <b>SDM2</b> | <b>MVI</b>  | <b>A,16</b> |
|             | <b>LXI</b>  | <b>H,0</b>  |
| <b>SDM3</b> | <b>DAD</b>  | <b>H</b>    |
|             | <b>RCL</b>  |             |
|             | <b>RBL</b>  |             |
|             | <b>BNC</b>  | <b>SDM4</b> |

negated by double subtraction from zero  
 subtract low order from zero  
 and replace  
**MVI** is used here instead of **ZAR** because **ZAR** would clear the carry bit  
 subtract high order from zero, propagate carry  
 and replace it  
 now test multiplicand sign  
 ...  
 skip if already positive since HL and DE can be exchanged we can negate the multiplicand by direct double subtraction. It is not necessary to clear carry before subtracting. The **ORA A** above left it zero.  
 form positive multiplicand  
 and replace it in DE  
 count to A  
 clear low product  
 shift low product and higher orders  
 and multiplier bit to carry  
 skip if multiplier bit zero

|             |            |             |                                                                                                                                                         |
|-------------|------------|-------------|---------------------------------------------------------------------------------------------------------------------------------------------------------|
|             | <b>DAD</b> | <b>D</b>    | add multiplicand                                                                                                                                        |
|             | <b>BNC</b> | <b>SDM4</b> | skip if no carry to third order                                                                                                                         |
|             | <b>INR</b> | <b>C</b>    | propagate carry to third order                                                                                                                          |
|             | <b>BNZ</b> | <b>SDM4</b> | skip if no carry to fourth order                                                                                                                        |
|             | <b>INR</b> | <b>B</b>    | propagate carry to fourth order                                                                                                                         |
| <b>SDM4</b> | <b>DCR</b> | <b>A</b>    | decrement step count                                                                                                                                    |
|             | <b>BNZ</b> | <b>SDM3</b> | and do it again if not zero                                                                                                                             |
|             | <b>LDA</b> | <b>SIGN</b> | now get sign indicator back                                                                                                                             |
|             | <b>ORA</b> | <b>A</b>    | test it for negative                                                                                                                                    |
|             | <b>JP</b>  | <b>SDM5</b> | skip if product positive                                                                                                                                |
|             | <b>MVI</b> | <b>D,0</b>  | zero is put into D so it can be moved to A as needed. This is more economical than the four <b>MVI A,0</b> instructions which would otherwise be needed |
|             | <b>ZAR</b> |             | zero A for first subtraction                                                                                                                            |
|             | <b>SUB</b> | <b>L</b>    | subtract low order from zero                                                                                                                            |
|             | <b>MOV</b> | <b>L,A</b>  | replace it                                                                                                                                              |
|             | <b>MOV</b> | <b>A,D</b>  | this again clears A but <i>not</i> carry                                                                                                                |
|             | <b>SBB</b> | <b>H</b>    | propagate carry                                                                                                                                         |
|             | <b>MOV</b> | <b>H,A</b>  | replace second order                                                                                                                                    |
|             | <b>MOV</b> | <b>A,D</b>  | clear A again, but not carry                                                                                                                            |
|             | <b>SBB</b> | <b>C</b>    | propagate carry to third order                                                                                                                          |



|             |            |            |                       |
|-------------|------------|------------|-----------------------|
|             | <b>MOV</b> | <b>C,A</b> | replace third order   |
|             | <b>MOV</b> | <b>A,D</b> | clear A again but not |
|             |            |            | carry                 |
|             | <b>SBB</b> | <b>B</b>   | propagate carry to    |
|             |            |            | highest order         |
|             | <b>MOV</b> | <b>B,A</b> | replace highest order |
| <b>SMD5</b> | <b>...</b> |            | program continues     |

As with addition and subtraction, the process of multiplication is indifferent to the placing of any point in the word. Scale factors are entirely the business of the programmer and if numbers with scale factors are multiplied the programmer must provide any shifting necessary to scale the result. If numbers with scale factors A and B are multiplied the scale factor of the product is A + B. Consider the multiplication of the two pure fractions:

$$.10000000_2 = .5_{10}$$

and

$$.11000000_2 = .75_{10}$$

If the points are ignored and the numbers read as integers they are  $128_{10}$  and  $192_{10}$ . If the product of these numbers is found by the method of example 8-2 it is:

$$0110000000000000_2$$

which is 24,576 if read as an integer. The scale factor of this product is the sum of the scale factors of the multiplier and multiplicand, i.e.,  $8 + 8 = 16$ . To reduce the product to the same scale factor as the multiplier and multiplicand it must be shifted right eight bits, yielding:

$$.01100000_2 = .375_{10}$$

Binary division is done by methods which parallel those used in pencil and paper decimal division. A double precision number known as the *dividend* is divided by a single precision *divisor* to yield

two single precision results, the *quotient* and the *remainder*. A constraint exists for division in a computer which does not exist for the pencil and paper division. The quotient must be expressible in a single computer word, eight bits in the case of the Z80. The condition in which the quotient cannot be held in a single computer word is known as the *divide fault* condition. We will return to divide fault shortly. It is sufficient to know here that the high order dividend, considered by itself as an integer, must be less in magnitude than the single precision divisor.

The mechanics of the division process are as follows. The double precision dividend is shifted left and the high half compared to the divisor. If the high half of the dividend is greater than or equal to the divisor, the divisor is subtracted from it and the low bit of the quotient set to one. If the high half of the dividend is less than the divisor no subtraction is performed and the low quotient bit is set to zero. The quotient is then rotated left. This process is repeated as many times as there are bits in the computer word, eight in this case.

As with multiplication, there are economies which can be made by combining certain of these functions. As the divisor is shifted left its vacated low bits can be used to hold the developing quotient, so that the quotient "grows" into the space vacated by the left shifting dividend. The simple division process is shown in example 8-8.

#### Example 8-8

A double precision dividend is in the HL register pair, with the single precision divisor being in D. Divide HL by D, leaving the quotient in L and the remainder in H.

| <i>label</i> | <i>inst.</i> | <i>operand</i> |                          |
|--------------|--------------|----------------|--------------------------|
| DV           | MVI          | B,8            | divide step count to B   |
|              | MVI          | E,0            | clear E. This is         |
|              |              |                | necessary because the    |
|              |              |                | DE pair will be used to  |
|              |              |                | subtract divisor from    |
|              |              |                | dividend                 |
| DV1          | DAD          | H              | shift divisor left. This |

|            |             |            |
|------------|-------------|------------|
|            | <b>DSC</b>  | <b>D</b>   |
|            | <b>BC</b>   | <b>DV2</b> |
|            | <b>INX</b>  | <b>H</b>   |
|            | <b>BRA</b>  | <b>DV3</b> |
| <b>DV2</b> | <b>DAD</b>  | <b>D</b>   |
| <b>DV3</b> | <b>DBNZ</b> | <b>DV1</b> |
|            | <b>...</b>  |            |

vacates the low bit of L which will be used to hold the quotient bit subtract divisor from high dividend. The subtraction is done directly without the compare. If the subtraction is unsuccessful it will be reversed by the **DAD D** instruction below if carry was set the high dividend was less than the divisor, i.e., the subtraction was unsuccessful if there was no carry the subtract was successful and the quotient bit is set to one skip restoration of high dividend restore high dividend decrement count and do it again if count not yet zero

We must now face the divide fault problem, and the constraints it places on dividend and divisor magnitudes. Obviously the divisor cannot be zero. Further, the high bit of the dividend must not be a one, since the **DAD H** would shift this bit out before the first divide step. Finally, if the high dividend is equal to or greater than the divisor, the divide cannot be attempted since a quotient greater than eight bits would be generated. This does *not* mean that larger quotients cannot be computed, just that it cannot be done with this sin-

gle precision division. A single precision division which incorporates the method of example 8-8 and checks for the divide fault condition is shown in example 8-9.

#### Example 8-9

Perform the same division as in example 8-8, but check for the divide fault condition. If there is a divide fault send control to location **DVDFLT**, otherwise perform the division.

| <i>label</i> | <i>inst.</i> | <i>operand</i> |                                                                                                           |
|--------------|--------------|----------------|-----------------------------------------------------------------------------------------------------------|
| <b>DVD</b>   | <b>MOV</b>   | <b>A,H</b>     | high dividend to A                                                                                        |
|              | <b>ORA</b>   | <b>A</b>       | set flags                                                                                                 |
|              | <b>JM</b>    | <b>DVDFLT</b>  | divide fault if high bit on                                                                               |
|              | <b>CMP</b>   | <b>D</b>       | compare high dividend to divisor                                                                          |
|              | <b>BGE</b>   | <b>DVDFLT</b>  | divide fault if high dividend greater than or equal to divisor. This also covers the case of zero divisor |
|              | <b>MVI</b>   | <b>B,8</b>     | perform division as before                                                                                |
|              | <b>MVI</b>   | <b>E,0</b>     | ...                                                                                                       |
| <b>DVD1</b>  | <b>DAD</b>   | <b>H</b>       | ...                                                                                                       |
|              | <b>DSC</b>   | <b>D</b>       |                                                                                                           |
|              | <b>BC</b>    | <b>DVD2</b>    |                                                                                                           |
|              | <b>INX</b>   | <b>H</b>       |                                                                                                           |
|              | <b>BRA</b>   | <b>DVD3</b>    |                                                                                                           |
| <b>DVD2</b>  | <b>DAD</b>   | <b>D</b>       |                                                                                                           |
| <b>DVD3</b>  | <b>DBNZ</b>  | <b>DVD1</b>    |                                                                                                           |
|              |              | <b>...</b>     |                                                                                                           |

Division of signed numbers is best done by forcing both dividend and divisor positive and adjusting the signs of quotient and remain-

der at the end, as was done with the multiplication in example 8-3. This will not be covered with an example here, since the mechanics are mostly repetition of parts of previous examples. Note here though that while the quotient sign can be determined by Exclusive ORing the signs of the divisor and dividend, the sign of the remainder is always the sign of the original dividend. This is to satisfy the condition:

$$(\text{divisor})(\text{quotient}) + \text{remainder} = \text{dividend}$$

Division of a 32 bit dividend by a 16 bit divisor involves exactly the same principles shown for single precision division above. Some special advantage can be taken here of the Z80's ability to shift any register. The process is shown in example 8-10.

#### Example 8-10

A quadruple precision dividend is in H, L, B and C, from highest order to lowest. A double precision divisor is in DE. Perform the indicated division leaving the quotient in BC and the remainder in HL. If there is a divide fault send control to **DVDFLT**.

| <i>label</i> | <i>inst.</i> | <i>operand</i> |                                     |
|--------------|--------------|----------------|-------------------------------------|
| <b>DDVD</b>  | <b>MOV</b>   | <b>A,H</b>     | highest order dividend to A         |
|              | <b>ORA</b>   | <b>A</b>       | set flags                           |
|              | <b>JM</b>    | <b>DVDFLT</b>  | cannot divide if high bit a one     |
|              | <b>SHLD</b>  | <b>HDVDND</b>  | store highest two dividend orders   |
|              | <b>SDED</b>  | <b>DVSR</b>    | store divisor                       |
|              | <b>EXX</b>   |                | use auxiliary registers for compare |
|              | <b>LHLD</b>  | <b>HDVDND</b>  | high two dividend orders to HL'     |
|              | <b>LDED</b>  | <b>DVSR</b>    | divisor to DE'. Note                |



|              |            |               |
|--------------|------------|---------------|
|              | <b>RSC</b> |               |
|              | <b>DSC</b> | <b>D</b>      |
|              | <b>BNC</b> | <b>DVDFLT</b> |
|              | <b>EXX</b> |               |
|              | <b>MVI</b> | <b>A,16</b>   |
| <b>DDVD1</b> | <b>LLC</b> |               |
|              | <b>RBL</b> |               |
|              | <b>DAC</b> | <b>H</b>      |
|              | <b>DSC</b> | <b>D</b>      |
|              | <b>BNC</b> | <b>DDVD2</b>  |
|              | <b>DAD</b> | <b>D</b>      |
|              | <b>BRA</b> | <b>DDVD3</b>  |
| <b>DDVD2</b> | <b>INR</b> | <b>C</b>      |
| <b>DDVD3</b> | <b>DCR</b> | <b>A</b>      |
|              | <b>BNZ</b> | <b>DDVD1</b>  |

...

that the **SHLD**, **SDED**, **LHLD** and **LDED** instructions used to move the high dividend and divisor here used 14 program words. There is a *much* better way to do this which uses only 4 words. This method will be discussed in chapter 9

clear carry for subtract  
subtract divisor from high dividend  
divisor less than high dividend if no carry, cannot do this division.  
get principal registers back  
divide step count to A  
shift dividend left

...  
this **DAC** brings any bit shifted out of B into the low bit of HL  
try the subtraction  
successful subtraction if no carry  
otherwise add it back  
and skip setting quotient bit  
set quotient bit to one  
decrement step count  
and do it again if not yet zero

Note when implementing this program that the quantities **HDVDND** and **DVSR** are *two* words long, i.e., they should be defined by:

```
HDVDND DBL 0
DVSR    DBL 0
```

Failing to remember this will lead to some mysterious errors.

True double precision multiplication and division, in which the numbers must be operated upon piecewise, is a bit more complicated. For multiplication both multiplier and multiplicand are considered to be two part numbers, the high and low orders having different scale factors. The product is then found from the following identity:

$$(A + B)(C + D) = AC + AD + BC + BD$$

Each of the four multiplications generates a double precision product. If the parts of each of these products are identified by suffixing H for high and L for low, the addition scheme for the partial products is:

| <i>4th order</i> | <i>3rd order</i> | <i>2nd order</i> | <i>1st order</i> |
|------------------|------------------|------------------|------------------|
| ACH              | ACL              |                  |                  |
|                  | ADH              | ADL              |                  |
|                  | BCH              | BCL              |                  |
|                  |                  | BDH              | BDL              |

If the numbers being multiplied are pure fraction the 1st and 2nd orders may be neglected in the interest of enhanced speed. This will tend to make the products too small in the low bit of the 3rd order since the carry propagated from the addition of the first two orders will be missing. For many purposes this will make no difference.

True double precision division usually arises in one of two different situations. In the first case a quotient more than eight bits long is

required from the division of a double precision dividend by a single precision divisor. This is the easy case. The first step is to perform a normal division, saving the quotient from this division as the high order quotient. The remainder from this division is then used as the high dividend for the next stage, the low dividend being cleared. This is guaranteed to be divisible, since the remainder is less than the divisor by definition. The quotient of this second division is then saved as the second order quotient. The remainder from the second division is then used as a high dividend for the third stage division, the low dividend again being cleared. The quotient of the third stage division becomes the third order quotient. This process can be carried out to any required length.

The second case in which a multiprecision dividend must be divided by a multiprecision divisor is somewhat more difficult. If the required precision exceeds two it will be simpler to simulate the effect of a normal division in the required length. For the double precision case divisor and dividend are represented as in the multiplication case, two part numbers with different scale factors. The quotient is given by the expression:

$$\frac{A+B}{C+D} = \frac{A+B}{C} - \frac{D(A+B)}{C^2} + \frac{D^2(A+B)}{C^3} \dots$$

It is easy to see that this expression converges fairly rapidly. The  $A+B$  being constant,  $1/C$  may be viewed as a coefficient of the first term. This amounts to division by a high order term. The  $D/C^2$  coefficient of the second term is a division of a lower order term by the square of a high order term. This is equivalent to multiplication by a low order term. The  $D^2/C^3$  is a multiplication by a fourth order term. As with any infinite series with terms of alternating sign, the error due to truncation is in the direction of the last included term. The three terms shown above will yield a quotient good to 29 bits, the last bit tending to be high because the last included term is positive. An illustration of this type of divide is somewhat beyond the scope of a book like this. If the reader is curious about the specific details of the process it will be instructive to study the listings of the mathematical packages which come with most minicomputers.

## 9 / USING THE STACK POINTER

“jack-in-the-box. A child’s toy, consisting of a box, out of which, when the lid is raised, a figure springs.”

Webster’s New Collegiate Dictionary

Extensive use has been made of the concept of a pointer in the previous chapters of this book. The first pointer introduced was the program counter, a 16 bit register which is used to fetch the next instruction to be executed. Further examples have illustrated the use of address pointers held in registers to fetch data from memory. It is the purpose of this chapter to illustrate the use of another type of pointer which makes possible yet another mode of memory access on the Z80. This pointer register, known as the *stack pointer*, is a 16 bit register which can be loaded from memory by the **LSPD** instruction and stored into memory by the **SSPD** instruction. Like the other pointer registers it can also be loaded in the immediate mode by **LXI**. The use of these instructions will be illustrated in the examples of this chapter and those which follow.

Memory access using the pointer registers discussed until now has been restricted to accessing one word at a time. The stack pointer is different in that reference to memory through it always addresses two words at a time. The stack pointer points to an area of memory designated by the programmer as the *stack*. The stack may not be randomly addressed, but must be accessed as a series of double word fetches or stores from ascending or to descending memory addresses. Data entered into the stack through the stack pointer are said to be *pushed* onto the stack, while data fetched from the stack through the stack pointer are said to be *popped* from the stack. A further difference between the stack pointer and the other pointers is that the stack pointer is incremented or decremented automatically with each stack access.

The access of data from the stack somewhat resembles accessing the trays in the spring loaded tray dispensers found in cafeterias. The stack of trays may be of any depth up to the limit of the capacity of the device. Starting with the tray dispenser empty, trays may be successively loaded into the device, each tray placed on top forcing the ones below it down. When the trays are withdrawn from the stack, however, only the last one entered may be accessed directly. Popping a tray from the stack causes the device to automatically reposition itself so that the tray below will be the next one accessed. The dispenser behaves in such a way that the trays are withdrawn (popped) in exactly the opposite order from that in which they were entered (pushed).

Data are pushed onto the stack by the **PUSH** instruction. The assembler allows this to be spelled either **PUSH** or **PSH**. The contents of the HL register pair are pushed onto the stack by:

#### **PUSH    H**

only the upper register of the pair being named in the operand. When this instruction is executed the following series of operations occurs:

- 1) the 16 bit stack pointer is decremented;
- 2) the contents of the higher order register (H) are stored into the memory location pointed to by SP;
- 3) the stack pointer is again decremented;
- 4) the contents of the lower order register (L) are stored into the memory location pointed to by SP.

Note that this stores the contents of HL in exactly the same form, low order occupying the lower memory address, as that specified for double precision numbers in chapter 7. The BC and DE register pairs and the X and Y registers are pushed onto the stack by

**PUSH    B**  
**PUSH    D**



**PUSH    X**  
**PUSH    Y**

The stack pointer register itself may not be pushed. The A and F registers are considered a register pair for stack operation purposes and have the collective name PSW (program status word) when referred to as a pair. Do not be misled by this awkward piece of nomenclature. The PSW consists of *two* words, A and F, with the F register being considered to be the low order register of the pair. The AF register pair is pushed onto the stack by:

**PUSH    PSW**

Data are retrieved from the stack by the **POP** instruction, using one of the operands already specified for **PUSH**. When **POP** is executed the following series of operations takes place:

- 1) the contents of the memory location pointed to by SP are loaded into the low order register;
- 2) SP is incremented;
- 3) the contents of the memory location pointed to by the incremented SP are loaded into the high order register;
- 4) SP is again incremented;

A very simple application of **PUSH** and **POP** is shown in example 9-1.

#### Example 9-1

Given that the stack pointer has been loaded elsewhere in the program, use **PUSH** and **POP** to copy the contents of the AF register pair to the AF' register pair.

| <i>label</i> | <i>inst.</i> | <i>operand</i> |
|--------------|--------------|----------------|
|--------------|--------------|----------------|

|             |            |                         |
|-------------|------------|-------------------------|
| <b>PUSH</b> | <b>PSW</b> | push AF pair onto stack |
| <b>XAF</b>  |            | set to access AF'       |
| <b>POP</b>  | <b>PSW</b> | load AF' from stack     |
| ...         |            |                         |

It should be understood here that **PUSH** and **POP** operations are simply special types of load and store instructions. Pushing a register pair onto the stack does not change the contents of the register pair. Popping memory locations into a register pair does not change the contents of the memory locations. A slightly more complex application of **PUSH** and **POP** is shown in example 9-2.

#### Example 9-2

Perform the same operation as in example 6-2, i.e., swap the contents of the BC and DE register pairs, but this time use only **PUSH** and **POP** instructions. The stack pointer has been loaded elsewhere.

| <i>label</i> | <i>inst.</i> | <i>operand</i> |                         |
|--------------|--------------|----------------|-------------------------|
| <b>SWAP</b>  | <b>PUSH</b>  | <b>B</b>       | push BC pair onto stack |
|              | <b>PUSH</b>  | <b>D</b>       | push DE pair onto stack |
|              | <b>POP</b>   | <b>B</b>       | pop former DE           |
|              |              |                | contents to BC          |
|              | <b>POP</b>   | <b>D</b>       | pop former BC contents  |
|              |              |                | to DE                   |
| ...          |              |                |                         |

The most important feature of example 9-2 is that the swap of the register pairs required only four memory words for instructions, plus four words of reusable stack space. The swap in example 6-2 required 16 words for instructions and four locations of dedicated memory space. Careful use of the stack allows significant economies to be made in memory use. Another instance in which using the stack

as temporary storage would save memory is in example 8-4. In this example the algebraic sign of the product of a pair of numbers was determined by an **XRA A**, this result being saved by the **STA SIGN** instruction which followed the **XRA A**. The result was later retrieved by an **LDA SIGN** instruction and the flags again set by an **ORA A**. If, instead of the **STA SIGN**, the AF pair had been saved by **PUSH PSW**, both the A register and the flags at the time of the **PUSH** could later have been retrieved by **POP PSW**, saving the **STA SIGN**, **LDA SIGN** and **ORA A** instructions, plus the temporary word **SIGN** itself. These instructions occupy eight memory words. The **PUSH PSW** and **POP PSW** which replace them occupy only two.

The last item pushed onto the stack is referred to as the top of the stack even though this item occupies the stack location with the lowest memory address. This is a confusing bit of nomenclature, so remember it to avoid future trouble. Data which are pushed onto the stack occupy successively lower memory addresses, while data popped from the stack are taken from successively higher memory addresses.

The stack can be used to access data for arithmetic purposes, though this should be done very carefully and NEVER attempted in a situation in which a program interrupt can occur. Just why this is so will have to wait until we cover the subject of interrupts, but it is so. Such a use of the stack and stack pointer is shown in example 9-3.

### Example 9-3

A 100 word block starts at location **LEON**. This block contains 50 signed double precision numbers. Use the stack pointer to retrieve the numbers and find their sum, leaving it in HL.

| <i>label</i> | <i>inst.</i> | <i>operand</i> |                                     |
|--------------|--------------|----------------|-------------------------------------|
|              | <b>MVI</b>   | <b>B,50</b>    | count to B register                 |
|              | <b>LXI</b>   | <b>SP,LEON</b> | address of beginning of block to SP |
|              | <b>LXI</b>   | <b>H,0</b>     | clear sum                           |

|            |             |            |                                                |
|------------|-------------|------------|------------------------------------------------|
| <b>SUM</b> | <b>POP</b>  | <b>D</b>   | pop double precision<br>number into DE         |
|            | <b>DAD</b>  | <b>D</b>   | add to sum in HL                               |
|            | <b>DBNZ</b> | <b>SUM</b> | decrement count and do<br>it again if not zero |
|            | ...         |            |                                                |

Again, the above technique should never be used in environments in which an interrupt can occur, and should never be used at all if the nonmaskable interrupt is connected.

Up until now we have shown only two ways in which the stack pointer can be loaded, either directly from memory by **LSPD** or in the immediate mode by **LXI**. The stack pointer may also communicate directly with the HL register pair and the X and Y registers. The contents of SP may be added to HL, X and Y, by **DAD SP**, **X+SP**, and **Y+SP**. SP can be loaded directly from HL, X or Y by **SPHL**, **SPX** and **SPY**. In addition the two words at the top of the stack, the item most recently pushed, may be exchanged with the contents of H, X or Y by the **XTHL**, **XTX** and **XTY** instructions. The uses of these functions will be illustrated shortly.

At this point the reader should pause and review his understanding of the material so far presented in this chapter. If the stack pointer use so far described is in any way fuzzy the remainder of the chapter will be incomprehensible. A thorough understanding of stack operations is necessary to understand not just this chapter but most of the remainder of the book.

The data storage and retrieval operations described so far form only a small fraction of stack pointer use in the Z80 and other microcomputers. Indeed the operations shown so far could have been achieved, albeit clumsily, by other means, as many of them were in earlier chapters. What comes next is a class of operations which can only be done efficiently by use of the stack.

Consider for a moment the true double precision multiplication described immediately after example 8-10. This operation required four multiplications and a series of additions to form the quadruple precision product. Using only the information developed until now,

this would have required four copies of the necessary multiplication routine. What is required in situations like this is some means of keeping one copy of the multiplication routine, jumping to it when a product is required, then returning control to a point in the program just after the jump to the multiplication routine. Jumping is no problem. The multiplication routine might be located anywhere in memory and could be jumped to from anywhere in memory. The problem is how to get back to the instruction which follows the jump to the multiplication routine.

This could be done using the **PCHL**, **PCX** or **PCY** instructions. These instructions load the program counter with the contents of HL, X and Y respectively. X, for example, could be loaded with the address to which control is to be returned and the instruction **PCX** added to the end of the multiplication routine. This would look like this:

|             |               |                             |
|-------------|---------------|-----------------------------|
| <b>LXI</b>  | <b>X,HERE</b> | address of <b>HERE</b> to X |
|             |               | register                    |
| <b>JMP</b>  | <b>DMPY</b>   | multiply routine of         |
|             |               | example 8-6                 |
| <b>HERE</b> | <b>...</b>    | control would return        |
|             |               | here if the last instruc-   |
|             |               | tion of <b>DMPY</b> were    |
|             |               | <b>PCX</b> .                |

This scheme is certainly possible and will work, but suffers from the defect that the multiplication routine is deprived of the use of one of the fundamental registers of the Z80. This does no harm in this case since the **DMPY** routine does not use X, but the method is clumsy and would be unworkable if the situation required the use of all available register space. There must be a better way.

The better way is provided by the Z80 hardware. Recall now the heavy emphasis made earlier on the fact that the program counter always points to the instruction following the one being executed. If the instruction being executed were a jump, the program counter would point to the instruction after the jump. PC thus contains exactly the address we are seeking. There exists in the Z80 a special kind of jump which captures the contents of the PC so that they may be used to return to the point immediately following the jump. This



special jump is known as a *call*. When a call instruction is executed, the following steps occur:

- 1) the program counter is pushed onto the stack;
- 2) the value of the operand of the call instruction is loaded into the program counter, forcing control to be transferred to that location. Instruction execution resumes at this new location.

For the multiplication routine of example 8-6 this would be written:

### **CALL DMPY**

The execution of **DMPY** would then proceed. To return to the location following the **CALL DMPY** another special instruction is used. This instruction causes the top item in the stack to be popped into the program counter, which forces control to that memory address. Since the top stack item was the old program counter, this amounts to a restoration of the program counter to the value it had when the **CALL** was executed, i.e., instruction execution resumes at the location following the **CALL**. This special instruction which pops the top of the stack into the program counter is known as a *return* instruction. It is written:

### **RET**

and must be the last executed instruction of **DMPY**. The **DMPY** routine now looks like this:

|              |            |              |
|--------------|------------|--------------|
| <b>DMPY</b>  | <b>MVI</b> | <b>A,16</b>  |
|              | <b>LXI</b> | <b>H,0</b>   |
| <b>DMPY1</b> | <b>DAD</b> | <b>H</b>     |
|              | <b>RCL</b> |              |
|              | <b>RBL</b> |              |
|              | <b>BNC</b> | <b>DMPY2</b> |
|              | <b>DAD</b> | <b>D</b>     |
|              | <b>BNC</b> | <b>DMPY2</b> |
|              | <b>INR</b> | <b>C</b>     |

```

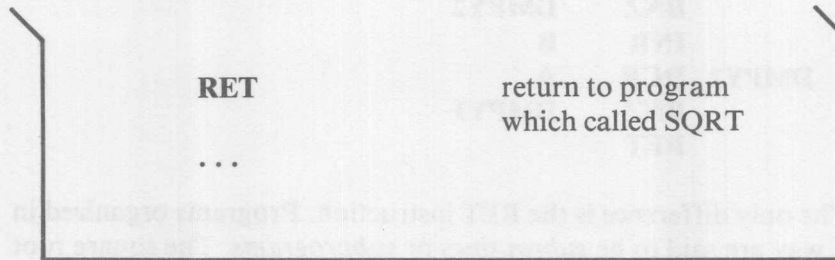
        BNZ    DMPY2
        INR    B
DMPY2   DCR    A
        BNZ    DMPY1
        RET
    
```

The only difference is the **RET** instruction. Programs organized in this way are said to be *subroutines* or *subprograms*. The square root program of example 4-1 can easily be rewritten as a subroutine, as shown in example 9-4.

#### Example 9-4

Using the method of example 4-1, construct a subroutine to extract the square root of the number in HL on entry to the subroutine.

| <i>label</i> | <i>inst.</i> | <i>operand</i> |                                                                                                                                                                                                                                               |
|--------------|--------------|----------------|-----------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------|
| <b>SQRT</b>  | <b>LXI</b>   | <b>D,1</b>     | this is the point to which control will be transferred by the <b>CALL</b> . It is known as the <i>entry point</i> of the subroutine and the label attached to this location, in this case <b>SQRT</b> , is called the <i>entry point name</i> |
|              |              |                | ...                                                                                                                                                                                                                                           |
| <b>LOOP</b>  | <b>RSC</b>   | <b>D</b>       | ...                                                                                                                                                                                                                                           |
|              | <b>DSC</b>   | <b>D</b>       | remainder of routine                                                                                                                                                                                                                          |
|              | <b>BC</b>    | <b>END</b>     | same as in example 4-1                                                                                                                                                                                                                        |
|              | <b>INX</b>   | <b>D</b>       | ...                                                                                                                                                                                                                                           |
|              | <b>INX</b>   | <b>D</b>       |                                                                                                                                                                                                                                               |
|              | <b>BRA</b>   | <b>LOOP</b>    |                                                                                                                                                                                                                                               |
| <b>END</b>   | <b>LRD</b>   |                |                                                                                                                                                                                                                                               |
|              | <b>RER</b>   |                |                                                                                                                                                                                                                                               |



The stack may be used for temporary storage within a subroutine in the same way as has been illustrated before. It is most important that the number of pushes be equal to the number of pops, however. If they are unequal the **RET** instruction at the end of the subroutine will pop something other than the old program counter value into PC and control will be transferred to somewhere other than the expected point. Just where control goes is dependent on exactly what was popped into PC by the **RET**. This leads to an unpredictable result which often results in large scale destruction of memory contents, making it very difficult to trace the error. Failure to equalize the number of pushes and pops in a subroutine is a very common stack programming error, second only to forgetting to load the stack pointer. It is an excellent programming practice to make loading the stack pointer the very first instruction of any program.

The choice of a memory area for use as a stack is up to the programmer. If you are very sure of the maximum stack size a fixed space in the program may be allocated as the stack area. If the stack should grow larger than the allocated space, however, the program or some of its operating constants may be destroyed, again leading to unpredictable results. A good practice is to use the area immediately below the program for stack space. Since the stack grows downward with successive pushes the chance for program damage is minimized this way. It can still occur, however, if a series of pops increments the stack pointer into the program area. The first push which follows will destroy program words. The debug monitor in appendix C contains a feature which will aid the programmer in controlling this error. This will be discussed in chapter 17.

The use of the stack area as temporary storage within a subroutine is illustrated in example 9-5.

## Example 9-5

A subroutine is required which will fill an area of memory with a constant value. The value to be stored into the block is in the A register on entry. The address of the first word of the block to be filled is in HL and the number of locations to be filled is in BC. Upon return from the subroutine all registers and flags are to be in exactly the same state as when the subroutine was called.

| <i>label</i> | <i>inst.</i> | <i>operand</i> |                                                                                                                                      |
|--------------|--------------|----------------|--------------------------------------------------------------------------------------------------------------------------------------|
| <b>FILL</b>  | <b>PUSH</b>  | <b>PSW</b>     | save A and F                                                                                                                         |
|              | <b>PUSH</b>  | <b>H</b>       | save HL                                                                                                                              |
|              | <b>PUSH</b>  | <b>B</b>       | save BC                                                                                                                              |
|              | <b>PUSH</b>  | <b>D</b>       | save DE. Since X and Y will not be used it is unnecessary to save them                                                               |
|              | <b>MOV</b>   | <b>D,H</b>     | copy HL to DE. This is necessary because we will use <b>MW+R</b> to fill the block. The "to" address must be in DE                   |
|              | <b>MOV</b>   | <b>E,L</b>     | now increment the "to" address. HL now points to the first word of the block and DE points to the second                             |
|              | <b>INX</b>   | <b>D</b>       | BC originally contained the total number of words to be filled. This is one more than the number of moves, so BC must be decremented |
|              | <b>DCX</b>   | <b>B</b>       | place value in first word                                                                                                            |
|              | <b>MOV</b>   | <b>M,A</b>     | fill the block                                                                                                                       |
|              | <b>MW+R</b>  |                |                                                                                                                                      |

|            |            |                 |
|------------|------------|-----------------|
| <b>POP</b> | <b>D</b>   | restore DE      |
| <b>POP</b> | <b>B</b>   | restore BC      |
| <b>POP</b> | <b>H</b>   | restore HL      |
| <b>POP</b> | <b>PSW</b> | restore A and F |
| <b>RET</b> |            |                 |

There are features of example 9-5 which require discussion before we leave it. The sharp-eyed reader may have noticed that the AF register pair was saved, but A was not changed. The flags were changed, however, which required saving AF. The subroutine was written as it is to illustrate something very important about the stack storage. If registers are to be restored to a previous state they must be popped in the exact opposite order from that in which they were pushed. Make sure you understand these operations very well before trying anything fancy with the stack. Remember, there is no such thing as a small error. Get a program that works first, by the simplest and most straightforward methods available. This way, if a disaster happens, you have something to which you can retreat.

We will leave example 9-5 with one final note. If the content of BC is zero on entry, the grandest of all program disasters will occur — nearly total memory destruction. It would be instructive for the reader to examine example 9-5 until he understands just what would happen and why.

The subroutine situations examined so far have all allowed the numbers to be operated upon, known as the *arguments* of the subroutine, to be communicated to the subroutine in the Z80's registers. This will not always be possible, either because the number of arguments is too large, or for other reasons. The problem is how to communicate the arguments without using the registers.

The solution to this problem is, in many ways, the most difficult thing you will encounter in this book. You may choose to skip over it, but you are urged not to do so. It is in the understanding of these difficult topics that your "sweat equity" in a computer is built up. If you stay with this and understand it your reward will be a real understanding of the Z80 stack structure and operations, allowing you to use the stack with ease and facility. This will pay off many times



in your future programming experience with the Z80 and other computers.

One solution to the problem is to store the arguments in a fixed area of memory. This gets around the stack problem entirely. The subroutine expects to find its arguments in fixed memory addresses and leaves its results in fixed memory addresses. This is sometimes a useful technique, but imposes an overhead on all the programs which may call the subroutine. The arguments must be moved into the fixed memory area, called the *common* area, with a series of loads and stores, and the results retrieved from the fixed locations and stored where required.

Another solution is to have the arguments themselves, the numbers to be operated upon, occupy the part of memory immediately following the **CALL** instruction. This results in a program which looks like this:

```
                CALL MUSIC
JOHN            ***
GEORGE         ***
PAUL           ***
RINGO          ***
                ...
```

The numbers **JOHN**, **GEORGE**, **PAUL** and **RINGO** are filled into the named locations by the calling program. The above sequence is known as the *calling sequence* for the subroutine. The way in which the arguments are retrieved is shown in example 9-6.

#### Example 9-6

A subroutine is required which will find the sum of the four numbers in the memory locations immediately following the **CALL** to the subroutine. This sum is to be returned in the A register. Control is to be returned to the location following the last argument (**RINGO+1** in the above calling sequence).

| <i>label</i> | <i>inst.</i> | <i>operand</i> |
|--------------|--------------|----------------|
| <b>MUSIC</b> | <b>XTHL</b>  |                |

|            |            |
|------------|------------|
| <b>MOV</b> | <b>A,M</b> |
| <b>INX</b> | <b>H</b>   |

|            |          |
|------------|----------|
| <b>ADD</b> | <b>M</b> |
|------------|----------|

|            |          |
|------------|----------|
| <b>INX</b> | <b>H</b> |
|------------|----------|

|            |          |
|------------|----------|
| <b>ADD</b> | <b>M</b> |
|------------|----------|

|            |          |
|------------|----------|
| <b>INX</b> | <b>H</b> |
|------------|----------|

|            |          |
|------------|----------|
| <b>ADD</b> | <b>M</b> |
|------------|----------|

swap HL with the top item on the stack. Since the last item pushed onto the stack was the program counter, HL now contains the memory address of the word which immediately follows the **CALL MUSIC** instruction, i.e., the address of **JOHN**. **JOHN** itself can be loaded directly into A by:

move **JOHN** to A

incrementing HL causes the register pair to point to the word following **JOHN**, i.e., to **GEORGE**, which can be added directly through HL

form sum of **JOHN + GEORGE**

increment HL to point to **PAUL**

form sum of **JOHN + GEORGE + PAUL**

increment HL to point to **RINGO**

and add **RINGO** to sum. We now have the final sum in A, but are faced with the problem of how to get back to the location which follows **RINGO**. HL can

**INX H**

**XTHL**

**RET**

be made to point to this location by:

control could be returned at this point by using **PCHL** to load the return address directly into PC, but this would leave the stack pointer changed. The problem is to get the desired return address into the stack so it can be popped into PC by the **RET** instruction. This can be accomplished by the same instruction used to get the original argument address from the stack, namely: swap HL with top of stack. HL has been restored to its original condition and the return address is at the top of the stack, ready to be popped by: return to **RINGO+1**

This sequence will work and produce correct results but is subject to the same objections as the common storage method, that the sequence must be filled in by the operating program. An even more serious limitation is that the method is useless for programs implemented in ROM, since the values of the arguments cannot be changed.

Yet another method is to communicate the address of the begin-

ning of a string of addresses of the arguments in a register pair. This involves a calling sequence that looks like this:

```

LXI  H,ARGSTR  address of string
                        of argument addresses
                        to HL
CALL SBRTNE    call the subroutine
  
```

with an argument address string, defined elsewhere, that looks like this:

```

ARGSTR  DBL  ARG1    address of first
                        argument
                        DBL  ARG2    address of second
                        argument
                        DBL  ARG3    address of third
                        argument
                        ...
                        etc.
  
```

A subroutine using this kind of calling sequence is shown in example 9-7.

#### Example 9-7

A series of signed Celcius temperatures for the seven days of the week is scattered in various locations in memory. A block containing their addresses is created by:

| <i>label</i>   | <i>inst.</i> | <i>operand</i> |                                         |
|----------------|--------------|----------------|-----------------------------------------|
| <b>TMP TRE</b> | <b>DBL</b>   | <b>SUNDAY</b>  | address of <b>SUNDAY</b><br>temperature |
|                | <b>DBL</b>   | <b>MONDAY</b>  | address of <b>MONDAY</b><br>temperature |
|                | <b>DBL</b>   | <b>TUESDY</b>  | address of <b>TUESDY</b><br>temperature |
|                | <b>DBL</b>   | <b>WDNSDY</b>  | etc.                                    |
|                | <b>DBL</b>   | <b>THRSDY</b>  | ...                                     |

```
DBL  FRIDAY  ...
DBL  SATRDY  ...
```

A subroutine is to be created, to be called by:

```
LXI  H,TMPTR
CALL LIMITS
...
```

The subroutine **LIMITS** is to examine the locations **SUNDAY** through **SATRDY** and return with the maximum temperature in B and the minimum in C.

|               |            |               |                                                                                                                                                                                                                                                                                                                                                                         |
|---------------|------------|---------------|-------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------|
| <b>LIMITS</b> | <b>MVI</b> | <b>B,-128</b> | set maximum register to most negative number                                                                                                                                                                                                                                                                                                                            |
|               | <b>MVI</b> | <b>C,127</b>  | and minimum register to most positive number. The testing will be done by comparing to these numbers, replacing the maximum with the value being examined if it is greater than the previous maximum and the minimum with the current value if it is less than the current minimum. Setting the registers this way insures at least one replacement. More on this below |
|               | <b>MOV</b> | <b>E,M</b>    | get low order address of <b>SUNDAY</b>                                                                                                                                                                                                                                                                                                                                  |
|               | <b>INX</b> | <b>H</b>      | bump pointer                                                                                                                                                                                                                                                                                                                                                            |
|               | <b>MOV</b> | <b>D,M</b>    | get high address of <b>SUNDAY</b>                                                                                                                                                                                                                                                                                                                                       |



|             |                 |             |                            |
|-------------|-----------------|-------------|----------------------------|
|             | <b>INX</b>      | <b>H</b>    | bump pointer               |
|             | <b>LDAX</b>     | <b>D</b>    | finally, get <b>SUNDAY</b> |
|             |                 |             | temperature in A           |
|             | <b>CMP</b>      | <b>B</b>    | compare to high            |
|             |                 |             | temperature                |
|             | <b>J &lt;</b>   | <b>LMT2</b> | skip if less               |
|             | <b>MOV</b>      | <b>B,A</b>  | new maximum                |
| <b>LMT2</b> | <b>CMP</b>      | <b>C</b>    | compare to low             |
|             |                 |             | temperature                |
|             | <b>J &gt; =</b> | <b>LMT3</b> | skip if not new low        |
|             | <b>MOV</b>      | <b>C,A</b>  | replace old low            |
|             |                 |             | temperature                |
| <b>LMT3</b> | <b>MOV</b>      | <b>E,M</b>  | get low address of         |
|             |                 |             | <b>MONDAY</b> , etc.       |

This process, beginning with the first **MOV E,M** will be repeated seven times. After these seven repetitions the maximum temperature for the week will be in B and the minimum in C. The only remaining instruction is:

**RET**

...

Such a subroutine has two features which draw the attention immediately. First, it is a task of repetitious mind-numbing tediousness to write and it uses a completely unreasonable amount of memory. It was presented this way so that the basic cycle of fetching the address, then fetching the temperature could be clearly seen. Now that this is understood, we can put the entire thing into a loop and produce a reasonable version. This is shown in example 9-8.

#### Example 9-8

Solve the same problem as in example 9-7, but use a loop to shorten the program.

| <i>label</i>  | <i>inst.</i>   | <i>operand</i> |                                                                                                                                                                                                   |
|---------------|----------------|----------------|---------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------|
| <b>LIMITS</b> | <b>MVI</b>     | <b>B,-128</b>  | load B and C with largest negative and positive numbers as before                                                                                                                                 |
|               | <b>MVI</b>     | <b>C,127</b>   |                                                                                                                                                                                                   |
|               | <b>EXX</b>     |                |                                                                                                                                                                                                   |
|               | <b>MVI</b>     | <b>B,7</b>     | swap register sets load count into B'. This is done so that the <b>DBNZ</b> instruction can be used. X or Y is not used because decrementing them sets no flags, making the test for zero awkward |
| <b>LMT1</b>   | <b>EXX</b>     |                | swap principal registers back                                                                                                                                                                     |
|               | <b>MOV</b>     | <b>E,M</b>     | get low order address                                                                                                                                                                             |
|               | <b>INX</b>     | <b>H</b>       | bump pointer                                                                                                                                                                                      |
|               | <b>MOV</b>     | <b>D,M</b>     | get high order address                                                                                                                                                                            |
|               | <b>INX</b>     | <b>H</b>       | bump pointer                                                                                                                                                                                      |
|               | <b>LDAX</b>    | <b>D</b>       | get temperature through pointer in DE                                                                                                                                                             |
|               | <b>CMP</b>     | <b>B</b>       | compare to high temperature                                                                                                                                                                       |
|               | <b>J &lt;</b>  | <b>LMT2</b>    | skip if less                                                                                                                                                                                      |
|               | <b>MOV</b>     | <b>B,A</b>     | otherwise replace with new maximum                                                                                                                                                                |
|               |                |                |                                                                                                                                                                                                   |
| <b>LMT2</b>   | <b>CMP</b>     | <b>C</b>       | compare to low temperature                                                                                                                                                                        |
|               | <b>J &gt;=</b> | <b>LMT3</b>    | skip if not new low                                                                                                                                                                               |
|               | <b>MOV</b>     | <b>C,A</b>     | otherwise replace old low                                                                                                                                                                         |
| <b>LMT3</b>   | <b>EXX</b>     |                | swap registers to get at count                                                                                                                                                                    |
|               | <b>DBNZ</b>    | <b>LMT1</b>    | repeat if count not yet zero                                                                                                                                                                      |

**EXX**

otherwise swap to  
access principal  
registers  
and then return

**RET**

The purpose of using the largest positive number as the initial minimum and the largest negative number as the initial maximum was to assure that at least one replacement of both would take place. Suppose the maximum and minimum had both been set at zero initially, and the temperature had varied between 10 and 20 degrees over the week. The maximum would have been recorded correctly since the first comparison would have replaced the zero, but the minimum would have gone unrecorded, since none of the temperatures would have been below zero. This would have given the false result of a zero minimum. The only safe beginning values are those which are sure to be replaced. Notice also that after the test for maximum is made the test for minimum is made even if the maximum test caused the old maximum value to be replaced. If the **MOV B,A** had been followed by a **BRA LMT3** the routine would have been faster, but would have failed in the circumstance in which the temperature increased each day. If you don't understand why this is so, it will be worth your while to work it out on paper until you do. The determination of maximum and minimum values in a list of numbers is an extremely common programming requirement and you should understand how it is done.

The final method for communication of subroutine arguments that we will consider is that in which the arguments addresses are in memory immediately after the **CALL** instruction. This leads to a calling sequence that looks like this:

|             |               |                         |
|-------------|---------------|-------------------------|
| <b>CALL</b> | <b>SUBRTN</b> |                         |
| <b>DBL</b>  | <b>ARG1</b>   | address of 1st argument |
| <b>DBL</b>  | <b>ARG2</b>   | address of 2nd          |
|             |               | argument                |
| <b>DBL</b>  | <b>ARG3</b>   | address of 3rd          |
|             |               | argument                |

...  
...  
**DBL      ARGn      address of nth  
                         argument**

The procedure for retrieving the arguments from this kind of calling sequence is necessarily a bit complicated, so it will be taken in two steps. The first step is the poorer solution, but is easier to understand and forms the base for understanding the second solution. The first solution is shown in example 9-9.

### Example 9-9

A subroutine is to be constructed which will examine the contents of three memory locations whose addresses are given in the calling sequence immediately after the **CALL** instruction. The largest of the three numbers is to be returned in the A register and its address in DE. Control should be returned to the point immediately following the last argument address.

| <i>label</i>  | <i>inst.</i> | <i>operand</i> |                            |
|---------------|--------------|----------------|----------------------------|
| <b>GRTEST</b> | <b>SSPD</b>  | <b>SPSAVE</b>  | save the stack pointer     |
|               | <b>XTHL</b>  |                | get the old program        |
|               |              |                | counter from the top of    |
|               |              |                | the stack. This is the ad- |
|               |              |                | dress of the first argu-   |
|               |              |                | ment address               |
|               | <b>SPHL</b>  |                | move the address to the    |
|               |              |                | stack pointer              |
|               | <b>POP</b>   | <b>H</b>       | address of 1st argument    |
|               |              |                | in H                       |
|               | <b>MOV</b>   | <b>D,H</b>     | copy address to DE         |
|               | <b>MOV</b>   | <b>E,L</b>     | ...                        |
|               | <b>MOV</b>   | <b>A,M</b>     | move argument to A         |
|               | <b>POP</b>   | <b>H</b>       | 2nd argument address       |
|               |              |                | to HL                      |

|            |             |               |                                                                                                                                                                                                                        |
|------------|-------------|---------------|------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------|
|            | <b>CMP</b>  | <b>M</b>      | compare 1st argument to 2nd                                                                                                                                                                                            |
|            | <b>BGE</b>  | <b>GR2</b>    | 1st argument greater than or equal to second, continue test                                                                                                                                                            |
|            | <b>MOV</b>  | <b>D,H</b>    | otherwise save address of 2nd argument                                                                                                                                                                                 |
|            | <b>MOV</b>  | <b>E,L</b>    | ...                                                                                                                                                                                                                    |
|            | <b>MOV</b>  | <b>A,M</b>    | and get 2nd argument to compare to 3rd                                                                                                                                                                                 |
| <b>GR2</b> | <b>POP</b>  | <b>H</b>      | 3rd argument address to HL                                                                                                                                                                                             |
|            | <b>CMP</b>  | <b>M</b>      | compare greater of first two arguments to third                                                                                                                                                                        |
|            | <b>BGE</b>  | <b>GR3</b>    | skip if A greater than 3rd argument                                                                                                                                                                                    |
|            | <b>MOV</b>  | <b>D,H</b>    | otherwise replace address                                                                                                                                                                                              |
|            | <b>MOV</b>  | <b>E,L</b>    | ...                                                                                                                                                                                                                    |
|            | <b>MOV</b>  | <b>A,M</b>    | and get 3rd argument                                                                                                                                                                                                   |
| <b>GR3</b> | <b>LXI</b>  | <b>H,0</b>    | at this point the largest of the three arguments is in A and its address is in DE. The return address must now be restored to the top of the stack and the stack pointer itself restored to accomplish a proper return |
|            | <b>DAD</b>  | <b>SP</b>     | adding SP to the cleared HL pair simply transfers SP to HL                                                                                                                                                             |
|            | <b>LSPD</b> | <b>SPSAVE</b> | restore the stack pointer to its condition on entry                                                                                                                                                                    |
|            | <b>XTHL</b> |               | place return address at top of stack                                                                                                                                                                                   |



**RET**

## and return

When we were finished popping the argument addresses into HL, the stack pointer contained the address of the word which followed the last argument address, i.e., the address of the desired return point. Restoring the original stack pointer and swapping HL with the top of the stack placed this address in the right position to be popped into the program counter by the **RET**.

Now, it should be clearly understood that this example was included purely for learning purposes. It will work and produce the claimed result, but like the earlier use of the stack pointer to pop double precision numbers in example 9-3, this technique should NEVER be used if an interrupt is possible, and NEVER be used at all if the nonmaskable interrupt is even connected. If the example is understood, however, it will pave the way to the understanding of a truly general method for argument retrieval in such calling sequences. This method is shown in example 9-10.

## Example 9-10

Solve the problem of example 9-9, but without the use of the stack pointer. All other registers may be used.

| <i>label</i>  | <i>inst.</i> | <i>operand</i> |
|---------------|--------------|----------------|
| <b>GRTEST</b> | <b>XTX</b>   |                |

|            |          |
|------------|----------|
| <b>LHX</b> | <b>1</b> |
| <b>LLX</b> | <b>0</b> |

swap top of stack with X register. X now points to the address of the 1st argument high address to H low argument address to L. These two instructions are taking the

|            |  |            |            |                                                                                                                                |
|------------|--|------------|------------|--------------------------------------------------------------------------------------------------------------------------------|
|            |  |            |            | place of the <b>POP H</b><br>which brought up the<br>argument address in ex-<br>ample 9-9                                      |
|            |  | <b>MOV</b> | <b>D,H</b> | save address, same as<br>before                                                                                                |
|            |  | <b>MOV</b> | <b>E,L</b> | ...                                                                                                                            |
|            |  | <b>MOV</b> | <b>A,M</b> | get 1st argument                                                                                                               |
|            |  | <b>LHX</b> | <b>3</b>   | get 2nd argument                                                                                                               |
|            |  | <b>LLX</b> | <b>2</b>   | ...                                                                                                                            |
|            |  | <b>CMP</b> | <b>M</b>   | compare 1st and 2nd<br>arguments                                                                                               |
|            |  | <b>BGE</b> | <b>GR2</b> | skip if 1st argument<br>greater than or equal to<br>second                                                                     |
|            |  | <b>MOV</b> | <b>D,H</b> | otherwise move 2nd<br>argument address to<br>DE                                                                                |
|            |  | <b>MOV</b> | <b>E,L</b> | ...                                                                                                                            |
|            |  | <b>MOV</b> | <b>A,M</b> | and get 2nd argument                                                                                                           |
| <b>GR2</b> |  | <b>LHX</b> | <b>5</b>   | get 3rd argument<br>address                                                                                                    |
|            |  | <b>LLX</b> | <b>4</b>   | ...                                                                                                                            |
|            |  | <b>CMP</b> | <b>M</b>   | compare 2nd argument<br>to 3rd                                                                                                 |
|            |  | <b>BGE</b> | <b>GR3</b> | skip if A greater than or<br>equal to 3rd argument                                                                             |
|            |  | <b>MOV</b> | <b>D,H</b> | otherwise copy 3rd<br>argument address to<br>DE                                                                                |
|            |  | <b>MOV</b> | <b>E,L</b> | ...                                                                                                                            |
|            |  | <b>MOV</b> | <b>A,M</b> | and get 3rd argument                                                                                                           |
| <b>GR3</b> |  | <b>LXI</b> | <b>B,6</b> | at this point the return<br>address in X must be<br>updated. This did not<br>occur automatically as<br>in 9-9 with the popping |

**X+BC**

**XTX**

**RET**

of the argument addresses. The return address must be computed by adding the length of the argument address string to it, i.e., 6 words for 3 addresses this adds the contents of BC to X place return address at top of stack and return

Like example 9-7, this procedure could become very tedious if there were a large number of arguments involved. The procedure can be changed to loop through any required number of arguments by incrementing X after each fetch made through it. This would amount to changing each

|            |            |
|------------|------------|
| <b>LHX</b> | <b>N+1</b> |
| <b>LLX</b> | <b>N</b>   |

where N is the specified displacement, to:

|            |          |
|------------|----------|
| <b>LLX</b> | <b>0</b> |
| <b>INX</b> | <b>X</b> |
| <b>LHX</b> | <b>0</b> |
| <b>INX</b> | <b>X</b> |

The count could now be kept in the B register and **DBNZ** used to go through the loop. The BC pair would no longer be required to compute the return address, since X would be updated to point to the next location after each fetch. If you have really understood the material in this and the preceding chapters you should now be able to do this for yourself.

Like the jump instructions, both **CALL** and **RET** can be made

conditional, testing the bits of the F register just as the conditional jumps do. The conditional call and return instructions are:

|            |            |                                                                                |
|------------|------------|--------------------------------------------------------------------------------|
| <b>CNZ</b> | <b>RNZ</b> | call or return if result was nonzero                                           |
| <b>CZ</b>  | <b>RZ</b>  | call or return if result was zero                                              |
| <b>CNC</b> | <b>RNC</b> | call or return if carry is zero                                                |
| <b>CC</b>  | <b>RC</b>  | call or return if carry is one                                                 |
| <b>CPO</b> | <b>RPO</b> | call or return if parity is odd                                                |
| <b>CPE</b> | <b>RPE</b> | call or return if parity is even                                               |
| <b>CNV</b> | <b>RNV</b> | call or return if no overflow. This instruction is identical to <b>CPO/RPO</b> |
| <b>CV</b>  | <b>RV</b>  | call or return if overflow. This instruction is identical to <b>CPE/RPE</b>    |
| <b>CP</b>  | <b>RP</b>  | call or return if result was positive                                          |
| <b>CM</b>  | <b>RM</b>  | call or return if result was negative                                          |

In addition to these conditional calls and returns, the assembler supports certain synonyms for use after compare instructions, i.e.:

|            |            |                                                          |
|------------|------------|----------------------------------------------------------|
| <b>CEQ</b> | <b>REQ</b> | call or return if equal. Assembles as <b>CZ/RZ</b>       |
| <b>CNE</b> | <b>RNE</b> | call or return if not equal. Assembles as <b>CNZ/RNZ</b> |

The following synonyms are used after comparison of unsigned eight bit magnitudes.

|            |            |                                                                                 |
|------------|------------|---------------------------------------------------------------------------------|
| <b>CLS</b> | <b>RLS</b> | call or return if A is<br>less. Assembles as<br><b>CC/RC</b>                    |
| <b>CGE</b> | <b>RGE</b> | call or return if A<br>greater than or equal.<br>Assembles as<br><b>CNC/RNC</b> |

The following synonyms are for use after comparison of signed two's complement numbers.

|                 |                 |                                                                            |
|-----------------|-----------------|----------------------------------------------------------------------------|
| <b>C &lt;</b>   | <b>R &lt;</b>   | call or return if A is<br>less. Assembles as<br><b>CM/RM</b>               |
| <b>C &gt; =</b> | <b>R &gt; =</b> | call or return if A<br>greater than or equal.<br>Assembles as <b>CP/RP</b> |

A final note on compatibility with the 8080. If programs are to be kept compatible they must use no Z80 only instructions, i.e., no instructions which are flagged by the assembler with a dollar (\$) to the left of the label field in the assembly listing. There is one area of incompatibility which cannot be detected by the assembler, however. This involves the common use of the P/V flag in the F register to indicate both parity and overflow. Arithmetic instructions on the 8080 set this flag to indicate parity, while it is used to indicate overflow on the Z80 by arithmetic instructions. If a program is using this flag as a parity indicator, after arithmetic instructions, the programmer must be careful to set the flag to indicate parity rather than the overflow state after the arithmetic instruction. This can be done by executing **ORA A** after the arithmetic operation and before the parity of the result is tested. Note here that single register increment and decrement qualify as arithmetic instructions for this purpose. Be careful!



## 10 / HANDLING ARRAYS AND TABLES

“Now the engine, from its capability of performing by itself all these purely material operations, spares intellectual labor, which may be more profitably employed. Thus the engine may be considered as a real manufactory of figures which will lend its aid to those many useful arts and sciences that depend on numbers. Again, who can foresee the consequences of such an invention?”

*Sketch of the Analytic Engine*  
Invented by Charles Babbage. 1842  
L. F. Menabrea

Many references have been made in previous chapters to blocks of data, groups of numbers occupying adjacent memory addresses which are connected by some common purpose. These blocks are known as *arrays*, and if they contain information which is to be looked up with the aid of some key they are called *tables*.

Space for an array may be reserved in the program by means of the **RES** (reserve) or **BSS** (block started by symbol) pseudo-ops. **RES** and **BSS** are identical in meaning. They may be preceded by a label and if so the label identifies the first word of the reserved block, the one with the lowest memory address. Examples are:

|             |            |                |                                                                                                                                             |
|-------------|------------|----------------|---------------------------------------------------------------------------------------------------------------------------------------------|
|             | <b>RES</b> | <b>400</b>     | reserve a block of 400 <sub>10</sub><br>locations                                                                                           |
|             | <b>RES</b> | <b>:100,-5</b> | reserve a block 100 <sub>16</sub><br>locations long filled<br>with the binary<br>equivalent of -5 <sub>10</sub><br>(11111011 <sub>2</sub> ) |
| <b>CUES</b> | <b>BSS</b> | <b>@40,'Q'</b> | reserve a block 40 <sub>8</sub><br>locations long, filled<br>with the ASCII code for<br>the letter Q                                        |

(11010001<sub>2</sub>). The label **CUES** will refer to the lowest memory address in the block.

If the second operand, called the fill value, is omitted, the block will simply be skipped over when the program is loaded, the contents remaining unchanged. The operands of an **RES** or **BSS** need not be literal numbers or characters. The sequence:

```
TIGER EQU 250
BLAKE EQU @77
NIGHT RES TIGER,BLAKE
```

will fill a 250<sub>10</sub> word block with the value 77<sub>8</sub>. Note that anything symbolic which appears in the operand field of **RES** or **BSS** must have been defined at a point in the program before its use as an operand. If the definition of **TIGER** or **BLAKE** had followed the definition of **NIGHT** above, the assembler would have flagged it as a **P** error, meaning post-defined. This is also true of the **ORG** pseudo-op.

Short arrays of ASCII characters can be created by the **ASC** pseudo-op. The array to be created is enclosed within apostrophes in the operand of the **ASC**, like this:

```
WOOPEE ASC 'FREE BEER!!!'
```

The operand field of the **ASC** may extend all the way to the end of the source line. If it overruns the operand field, the assembler will move the comment field to the right to accommodate it. Arrays of ASCII character of any length may be created by use of successive **ASC** pseudo-ops, e.g.:

```
ASC 'IF THIS IS A WRONG NUMBER'
ASC ' WHY DID YOU ANSWER THE PHONE?'
```

Note when doing this that if the line is broken at a boundary between words, the blank between the words is not supplied automatically. The second **ASC** above begins with a blank for this reason.

Perhaps the simplest of all operations on an array is to search it for some value. One way of doing this is shown in example 10-1.

### Example 10-1

An array of ASCII characters has been created by:

| <i>label</i> | <i>inst.</i> | <i>operand</i>    |
|--------------|--------------|-------------------|
| <b>OSCAR</b> | <b>ASC</b>   | <b>'VTGQHKAP'</b> |

Create a subroutine to search this array for the character in the A register on entry. If the character is found, return with carry off and the ordinal position of the character in the array in C. If the character is not found in the array return with carry on.

|               |             |                |                                                                      |
|---------------|-------------|----------------|----------------------------------------------------------------------|
| <b>CSRCH</b>  | <b>LXI</b>  | <b>H,OSCAR</b> | address of <b>OSCAR</b> to<br>HL pair                                |
|               | <b>MVI</b>  | <b>B,8</b>     | count of characters to B                                             |
|               | <b>MVI</b>  | <b>C,1</b>     | 1 to C. This will be<br>incremented for each<br>unsuccessful compare |
| <b>CSRCH2</b> | <b>CMP</b>  | <b>M</b>       | compare character in A<br>to memory                                  |
|               | <b>BEQ</b>  | <b>GOTCHA</b>  | skip out of loop if<br>found                                         |
|               | <b>INX</b>  | <b>H</b>       | otherwise increment<br>pointer                                       |
|               | <b>INR</b>  | <b>C</b>       | and ordinal array<br>number                                          |
|               | <b>DBNZ</b> | <b>CSRCH2</b>  | decrement count and do<br>it again                                   |
|               | <b>STC</b>  |                | character not in<br><b>OSCAR</b> array, turn<br>carry on.            |
|               | <b>RET</b>  |                | and return                                                           |
| <b>GOTCHA</b> | <b>RSC</b>  |                | character found, clear                                               |

**RET**

carry  
return with ordinal  
number in C

...

Example 10-1 could have been made more efficient in a number of ways. The most obvious economy is in replacing the second and third instructions with a single instruction, **LXI B,:801**, which would have loaded both registers at once, saving a word of memory. The most significant economy, however, could have been made by using a special Z80 facility, the **CP+** and **CP+R** instructions. The instructions function somewhat like the **MW+** and **MW+R** instructions described in chapter 6. The word in A is compared to the memory word, the HL pointer incremented, and the BC counter decremented. In the case of **CP+** the flag bits are set as for an ordinary compare, excepting the overflow flag. If the word in A was equal to the word in memory the Z bit will be set and tested with a **BEQ** or **BNE** instruction. The overflow flag will be on if the execution of the **CP+** did not cause the count in BC to become zero, and off if it did become zero. The use of **CP+** is shown in example 10-2.

#### Example 10-2

Search the array **OSCAR** as in example 10-1, this time using the **CP+** instruction.

| <i>label</i>  | <i>inst.</i> | <i>operand</i> |                               |
|---------------|--------------|----------------|-------------------------------|
| <b>ASRCH</b>  | <b>LXI</b>   | <b>H,OSCAR</b> | array address to HL           |
|               | <b>LXI</b>   | <b>B,8</b>     | count to BC                   |
| <b>ASRCH2</b> | <b>CP+</b>   |                | perform the comparison        |
|               | <b>BEQ</b>   | <b>EUREKA</b>  | skip if equal compare         |
|               | <b>JV</b>    | <b>ASRCH2</b>  | count not zero if overflow on |

|               |                 |                                                                                                                                                                                       |
|---------------|-----------------|---------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------|
|               | <b>STC</b>      | set carry, character not found                                                                                                                                                        |
|               | <b>RET</b>      |                                                                                                                                                                                       |
| <b>EUREKA</b> | <b>PUSH PSW</b> | save character in A                                                                                                                                                                   |
|               | <b>MOV A,C</b>  | move low count to A.                                                                                                                                                                  |
|               |                 | This is a number between 0 and 7, 7 if the character was V and 0 if it was P, exactly the opposite of what is required. The 0-7 range is because BC was decremented by the <b>CP+</b> |
|               | <b>SUI 8</b>    | subtract 8. This creates a number between -1 and -8, -1 if the character was V and -8 if it was P                                                                                     |
|               | <b>NEG</b>      | negate the number. This generates a number from 1 to 8 as required                                                                                                                    |
|               | <b>MOV C,A</b>  | move number to C                                                                                                                                                                      |
|               | <b>POP PSW</b>  | restore AF                                                                                                                                                                            |
|               | <b>RSC</b>      | clear carry for successful search                                                                                                                                                     |
|               | <b>RET</b>      | return to calling program                                                                                                                                                             |

An even more efficient way to do the job is to use the repeating form of **CP+**, **CP+R**. When **CP+R** is executed the program counter is automatically decremented after each step in which BC has not become zero. This makes an elegant way to search an array for some value. The use of **CP+R** is shown in example 10-3.



## Example 10-3

Again solve the problem of example 10-1, but use the **CP+R** instruction.

| <i>label</i> | <i>inst.</i> | <i>operand</i> |                                                                                                                                                                                                                                 |
|--------------|--------------|----------------|---------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------|
| <b>SRCH</b>  | <b>LXI</b>   | <b>H,OSCAR</b> | address of <b>OSCAR</b><br>to HL                                                                                                                                                                                                |
|              | <b>PUSH</b>  | <b>H</b>       | push onto stack                                                                                                                                                                                                                 |
|              | <b>POP</b>   | <b>D</b>       | copy to DE. This time<br>we will generate the po-<br>sitional number in the<br>array a different way                                                                                                                            |
|              | <b>LXI</b>   | <b>B,8</b>     | count to BC, as before                                                                                                                                                                                                          |
| <b>SRCH2</b> | <b>CP+R</b>  |                | search the array. The<br><b>CP+R</b> will stop when<br>an equal compare has<br>been found or when the<br>count has been exhaust-<br>ed. If the count is ex-<br>hausted first the over-<br>flow flag is set as for<br><b>CP+</b> |
|              | <b>BEQ</b>   | <b>FOUND</b>   | skip if equal compare.<br>If control gets beyond<br>the <b>BEQ</b> the count went<br>to zero. It is therefore<br>unnecessary to check<br>overflow                                                                               |
|              | <b>STC</b>   |                | carry on for<br>unsuccessful search<br>and return                                                                                                                                                                               |
|              | <b>RET</b>   |                |                                                                                                                                                                                                                                 |
| <b>FOUND</b> | <b>RSC</b>   |                | clear carry for subtract                                                                                                                                                                                                        |
|              | <b>LXI</b>   | <b>D,OSCAR</b> | array address to DE                                                                                                                                                                                                             |
|              | <b>DSC</b>   | <b>D</b>       | this forms the<br>difference between the<br>address of the first ar-                                                                                                                                                            |

```

MOV    C,L

```

```

RET

```

ray member and the address of the matching character plus one, since the **CP+R** automatically incremented HL one past the matching character move number to C as required. Carry is already zero, since the **DSC** cannot have caused it to be set in these circumstances return to calling program

Most often the table to be searched will not be composed of single element entries as in the above examples, nor will the requirement be to return the simple table position at which the entry is found. What is usually required is that a table of multiple element entries be searched and the corresponding multiple element entry in a parallel table be returned. This situation is shown in example 10-4.

#### Example 10-4

Two parallel tables exist in memory. The first consists of four character names of products. This table originates at **PRTBLE**. The second table, beginning at **DTBL**, consists of two word entries, the first being the product class and the second the price. The name to be searched for is in locations **PRNAME** through **PRNAME + 3**. Search **PRTBL** for the name given in **PRNAME** through **PRNAME + 3**. If the item is found return its price in H and product class in L. If the item is not found return with **FFFF<sub>16</sub>** in H and L. The tables look like this:

| <i>label</i> | <i>inst.</i> | <i>operand</i> |
|--------------|--------------|----------------|
| <b>PRTBL</b> | <b>ASC</b>   | <b>'MILK'</b>  |
|              | <b>ASC</b>   | <b>'EGGS'</b>  |
|              | <b>ASC</b>   | <b>'BEER'</b>  |
|              | <b>ASC</b>   | <b>'MEAT'</b>  |
|              | <b>ASC</b>   | <b>'CORN'</b>  |

etc.

|             |             |              |                             |
|-------------|-------------|--------------|-----------------------------|
| <b>DTBL</b> | <b>DATA</b> | <b>0,40</b>  | class 0, 40 cents per quart |
|             | <b>DATA</b> | <b>5,68</b>  | class 5, 68 cents a dozen   |
|             | <b>DATA</b> | <b>3,88</b>  | class 3, 88 cents a quart   |
|             | <b>DATA</b> | <b>2,169</b> | class 2, \$1.69 a pound     |
|             | <b>DATA</b> | <b>9,55</b>  | class 9, 55 cents a can     |

etc.

There are 72 different items in the table, so we define **TBLSIZ**:

**TBLSIZ EQU 72**

and the table search routine:

|               |             |                 |                                        |
|---------------|-------------|-----------------|----------------------------------------|
| <b>PRSRC</b>  | <b>MVI</b>  | <b>B,TBLSIZ</b> | table size to B                        |
|               | <b>LXI</b>  | <b>H,PRTBL</b>  | name table origin to HL                |
| <b>PRSRC1</b> | <b>LXI</b>  | <b>D,PRNAME</b> | address of name to be found to DE      |
|               | <b>LDAX</b> | <b>D</b>        | first character of name to A           |
|               | <b>SUB</b>  | <b>M</b>        | subtract first character of table name |
|               | <b>INX</b>  | <b>D</b>        | increment both address pointers        |
|               | <b>INX</b>  | <b>H</b>        | ...                                    |

|                    |                             |
|--------------------|-----------------------------|
| <b>MOV C,A</b>     | and save difference in C    |
| <b>LDAX D</b>      | get second name             |
|                    | character                   |
| <b>SUB M</b>       | subtract second             |
|                    | character of table name     |
| <b>INX D</b>       | increment both address      |
|                    | pointers                    |
| <b>INX H</b>       | ...                         |
| <b>ORA C</b>       | OR with former result       |
| <b>MOV C,A</b>     | and save again in C         |
| <b>LDAX D</b>      | get third name              |
|                    | character                   |
| <b>SUB M</b>       | subtract third table        |
|                    | name character              |
| <b>INX D</b>       | increment both pointers     |
| <b>INX H</b>       | ...                         |
| <b>ORA C</b>       | OR again                    |
| <b>MOV C,A</b>     | save in C                   |
| <b>LDAX D</b>      | get last name character     |
| <b>SUB M</b>       | subtract last table name    |
|                    | character                   |
| <b>INX D</b>       | increment pointers          |
| <b>INX H</b>       | ...                         |
| <b>ORA C</b>       | OR for final result. If     |
|                    | the result of this OR is    |
|                    | zero, all four subtrac-     |
|                    | tions were zero and the     |
|                    | names were identical.       |
|                    | Had addition been used      |
|                    | instead, the names          |
|                    | <b>MEAT</b> and <b>MAET</b> |
|                    | would have compared         |
|                    | identical                   |
| <b>BZ WOOPEE</b>   | found it if zero here       |
| <b>DBNZ PRSRC1</b> | do it again if not last     |
|                    | table entry                 |

|                   |                |                                                                                                                                                                                                                                                                                                                                                                                                               |
|-------------------|----------------|---------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------|
| <b>LXI</b>        | <b>H,-1</b>    | if control reaches here<br>the name was not in the<br>table                                                                                                                                                                                                                                                                                                                                                   |
| <b>RET</b>        |                | return with fail code in<br>HL                                                                                                                                                                                                                                                                                                                                                                                |
| <b>WOOPEE LXI</b> | <b>D,PRTBL</b> | address of table origin<br>to DE                                                                                                                                                                                                                                                                                                                                                                              |
| <b>DSC</b>        | <b>D</b>       | form difference<br>between address of<br>name and table origin.<br>Note that this number<br>will be a multiple of 4<br>since each <b>PRTBL</b> entry<br>was 4 characters long.<br>Note also that it will be<br>four too large since the<br>pointers were incre-<br>mented before the final<br>test. It was unnecessary<br>to clear carry before<br>subtracting. This was<br>done by the <b>ORA C</b><br>above |
| <b>LRH</b>        |                | divide HL by two by<br>shifting right                                                                                                                                                                                                                                                                                                                                                                         |
| <b>RLR</b>        |                | we now have a table<br>displacement in HL<br>suitable for use with<br><b>DTBL</b> . It is still two<br>greater than the re-<br>quired value for the<br>same reason that the<br>computed displacement<br>above was four too big.<br>We could fix this by<br>decrementing twice, but<br>there is a better way                                                                                                   |



|             |                 |                                                              |
|-------------|-----------------|--------------------------------------------------------------|
| <b>LXI</b>  | <b>D,DTBL-2</b> | address of word two<br>below the beginning of<br><b>DTBL</b> |
| <b>DAD</b>  | <b>D</b>        | add this to the<br>displacement in HL                        |
| <b>MOV</b>  | <b>E,M</b>      | product class E                                              |
| <b>INX</b>  | <b>H</b>        | bump pointer                                                 |
| <b>MOV</b>  | <b>D,M</b>      | price to D                                                   |
| <b>XCHG</b> |                 | product class to L, price<br>to H                            |
| <b>RET</b>  |                 | and return to calling<br>program                             |
| <b>...</b>  |                 |                                                              |

The method of example 10-4 is very important. Be sure you understand what went on here before proceeding. The computation of displacements in this way is extremely common. The final touch can be put on this table search by getting rid of all the repetition involved in fetching the names from memory. This is done by having all of the fetching and comparison done in a subroutine. This is shown in example 10-5.

#### Example 10-5

Solve the same problem as in example 10-4, but put the data fetching and comparison into a loop to cut down memory use.

| <i>label</i>  | <i>inst.</i> | <i>operand</i>  |                                          |
|---------------|--------------|-----------------|------------------------------------------|
| <b>PRSRC</b>  | <b>MVI</b>   | <b>B,TBLSIZ</b> | prolog same as before                    |
|               | <b>LXI</b>   | <b>H,PRTBL</b>  | ...                                      |
| <b>PRSRC1</b> | <b>CALL</b>  | <b>CMPARE</b>   | call comparison<br>subroutine            |
|               | <b>BZ</b>    | <b>WOOPEE</b>   | return from subroutine<br>with flags set |

```

DBNZ PRSRC1    as before
LXI   H,-1
RET

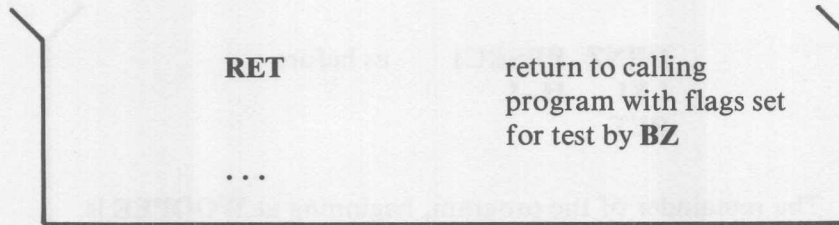
```

The remainder of the program, beginning at **WOOPEE** is the same. The comparison subroutine poses an interesting problem, however. We would like to use the B register to count the four comparisons, but it is already in use to count through the main table. The solution is as follows:

```

CMPARE  PUSH  B           save main program's
                           BC register pair
                           LXI   B,:400      load B with 4 and C
                                           with 0
                           LXI   D,PRNAME    address of name to DE.
                                           The table pointer is
                                           already in HL
CM1     LDAX  D           get name character
        SUB  M           subtract table character
        ORA  C           OR with result
        MOV  C,A         save new result
        INX  D           increment both pointers
        INX  H           ...
        DBNZ CM1        decrement count
        POP  B           restore main program
                           BC. Neither the POP,
                           the DBNZ nor the
                           INX's has any effect on
                           the flags. The flags
                           therefore are in the
                           state set by the
                           ORA  C. Since this is
                           the final test of the four
                           it will be zero if all of
                           the subtractions yielded
                           zero

```



A final refinement can be put on the method by abolishing the separate tables. Instead, a single table containing alternate name and code entries is established. This table has the form:

```
ASC  'MILK'
DATA 0,40
ASC  'EGGS'
DATA 5,68
```

and so forth. The search of such a table differs from those already shown only in that the table pointer must be incremented twice more after each unsuccessful compare to make it point to the next table entry. We will not bother with a formal example of this here, but the reader is urged to look in the source listing of the assembly program at location **OPSRC**. This is the program which searches the Z80 mnemonic table for the instruction the programmer has written. Each of the four word mnemonic entries is followed by two words specifying the instruction class, i.e., what kind of processing is required, and the binary skeleton of the instruction. Keep in mind that this and all other parts of the assembler use only the 8080 instruction subset. It would be an instructive exercise for the reader to rewrite this **OPSRC** routine to take advantage of economies possible with the added Z80 instructions.

Another common operation performed on arrays is known as sorting, the rearrangement of the members of the array in the smallest to largest or largest to smallest order. A vast amount of effort and ingenuity has gone into finding efficient solutions to this problem, occurring, as it does, so commonly in commercial applications. We will illustrate a very simple sorting method here known as a *bubble sort*. In a bubble sort adjacent array members are compared and switched if the comparison condition is met. If the array is being

sorted from smallest to largest the condition is that array member  $N$  is larger than array member  $N+1$ . The contents of  $N$  are swapped with the contents of  $N+1$  and the next comparison, between  $N+1$  and  $N+2$  is made, the same process being repeated. In this way the largest member of the array is "floated" to the top in the first pass. There are a couple of things about this process which may not be obvious on the face. First, for an array of  $K$  members, there are  $K-1$  comparisons to be done on each pass, and second, after each pass through the array the number of comparisons decreases by one since each pass "floats" the largest remaining number to the top. At the beginning of each pass a counter is set to zero. Each time a swap is made during the pass this counter is incremented. At the end of each sorting pass the counter is tested. When an entire pass through the array is made with no swaps the array is sorted. A very simple sort of this kind is shown in example 10-6.

#### Example 10-6

A bubble sort subroutine is required which will sort an array into order so that the smallest member occupies the lowest memory address. The numbers in the array are unsigned eight bit magnitudes. The address of the beginning of the array is held in HL on entry and the size of the array is in B. Return with carry on if the initial size of the array is given as zero, otherwise sort the array and return with carry off.

Advantage can be taken here of the Z80's indexed addressing capability. If X contains the address of an array member, that member can be loaded into A by **LAX 0** and compared to the following array member by **CMPX 1**. This eliminates the necessity for keeping and incrementing two pointers. The subroutine is shown below.

| <i>label</i> | <i>inst.</i> | <i>operand</i> |                                |
|--------------|--------------|----------------|--------------------------------|
| <b>START</b> | <b>ZAR</b>   |                | clear A for test of array size |

|              |             |              |                                                                                                                                                                                                                                         |
|--------------|-------------|--------------|-----------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------|
|              | <b>ORA</b>  | <b>B</b>     | set flags                                                                                                                                                                                                                               |
|              | <b>BZ</b>   | <b>SORT5</b> | array size zero, return with carry on                                                                                                                                                                                                   |
|              | <b>MVI</b>  | <b>C,0</b>   | clear swap count in C                                                                                                                                                                                                                   |
| <b>SORT1</b> | <b>PUSH</b> | <b>H</b>     | transfer array address from HL to X                                                                                                                                                                                                     |
|              | <b>POP</b>  | <b>X</b>     |                                                                                                                                                                                                                                         |
|              | <b>DCR</b>  | <b>B</b>     | decrement comparison counter. On the first pass this adjusts the count to K-1, where K is the array size. On succeeding passes it decrements to allow for the fact that the largest member found in the last pass is already at the top |
|              | <b>BZ</b>   | <b>SORT4</b> | the number of comparisons to be made is less than or equal to the number of swaps, by definition. If this number becomes zero the array is completely sorted                                                                            |
|              | <b>PUSH</b> | <b>B</b>     | save B and C                                                                                                                                                                                                                            |
| <b>SORT2</b> | <b>LAX</b>  | <b>1</b>     | get N+1 array member                                                                                                                                                                                                                    |
|              | <b>CMPX</b> | <b>0</b>     | compare to Nth array member                                                                                                                                                                                                             |
|              | <b>BGE</b>  | <b>SORT3</b> | skip if Nth less than N+1st                                                                                                                                                                                                             |
|              | <b>LDX</b>  | <b>0</b>     | otherwise swap Nth and N+1st                                                                                                                                                                                                            |
|              | <b>SAX</b>  | <b>0</b>     | ...                                                                                                                                                                                                                                     |
|              | <b>SDX</b>  | <b>1</b>     | ...                                                                                                                                                                                                                                     |



|              |             |              |                                                                                                                                                                                                                                                     |
|--------------|-------------|--------------|-----------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------|
|              | <b>INR</b>  | <b>C</b>     | increment swap counter                                                                                                                                                                                                                              |
| <b>SORT3</b> | <b>INX</b>  | <b>X</b>     | increment array pointer                                                                                                                                                                                                                             |
|              | <b>DBNZ</b> | <b>SORT2</b> | decrement count and repeat                                                                                                                                                                                                                          |
|              | <b>ZAR</b>  |              | clear A for test for swap count                                                                                                                                                                                                                     |
|              | <b>ORA</b>  | <b>C</b>     | test swap counter                                                                                                                                                                                                                                   |
|              | <b>POP</b>  | <b>B</b>     | this retrieves the comparison count from the stack <i>and</i> clears C for the next pass through the array, since C was cleared before BC was pushed above. This <b>POP</b> did not change any of the flags. They are still as they were set by the |
|              | <b>BZ</b>   | <b>SORT4</b> | <b>ORA C</b><br>skip if swap counter zero                                                                                                                                                                                                           |
| <b>SORT4</b> | <b>BRA</b>  | <b>SORT1</b> | otherwise do it again                                                                                                                                                                                                                               |
|              | <b>RSC</b>  |              | clear carry for normal return                                                                                                                                                                                                                       |
|              | <b>RET</b>  |              | return to calling program                                                                                                                                                                                                                           |
| <b>SORT5</b> | <b>STC</b>  |              | set carry for zero array size                                                                                                                                                                                                                       |
|              | <b>RET</b>  |              | return to calling program                                                                                                                                                                                                                           |
|              | <b>...</b>  |              |                                                                                                                                                                                                                                                     |

Without the use of the index plus displacement addressing facility of the Z80 this program would have been much longer. To see just how much longer the reader may consult example 9-4 of the author's earlier *Practical Microcomputer Programming: The Intel 8080* (Northern Technology Books, 1976).

One of the most common sorting requirements is for the sorting of a table of multilength character data into alphabetic order. The same principles apply to this kind of sort as to the simple sort we have just seen, but the mechanics get a little bit more complicated. A requirement like this arises when writing assembly programs. Each symbol used as a label is entered into a symbol table, which is printed at the end of the assembly. To be useful as a program reference it must be put into alphabetic order. If the reader has need of such techniques he is urged to examine the **SRTST** routine in the assembler listing. This routine uses a simple bubble sort of the type illustrated here. This is efficient enough for the relatively small numbers of symbols that have to be sorted by an assembler. If large amounts of alphabetic data are to be sorted, however, other techniques than the bubble sort should be examined. Descriptions of these techniques are available in a variety of places, in particular consult Flores' *Computer Sorting* (Prentiss-Hall, 1969).

The final table lookup technique to be considered here is that in which the incoming argument can be used to directly calculate the table position of the required quantity. This is shown in example 10-7.

#### Example 10-7

A subroutine is required which will transform a voltage read from a thermistor to a temperature. The thermistor voltage will be represented by a binary number between -20 and +35. The number representing voltage is to be used as a displacement to look up the corresponding temperature in a table which begins at **DGREEZ**. If the number representing voltage is in the range -20 to +35 look up the temperature and return it in A with carry off. If the number representing voltage is out of this range return with carry on. The incoming number representing voltage is in the A register.

| <i>label</i>  | <i>inst.</i>  | <i>operand</i> |                         |
|---------------|---------------|----------------|-------------------------|
| <b>THMSTR</b> | <b>CPI</b>    | <b>36</b>      | compare for upper limit |
|               | <b>J&gt;=</b> | <b>BUMMER</b>  | jump must be used       |

|               |                 |                                                                |
|---------------|-----------------|----------------------------------------------------------------|
| <b>CPI</b>    | <b>-20</b>      | because there are no<br>branch instructions                    |
| <b>J&lt;</b>  | <b>BUMMER</b>   | which test the sign flag<br>compare for lower limit            |
| <b>ADI</b>    | <b>20</b>       | below bottom of range<br>normalize range to zero               |
| <b>MOV</b>    | <b>E,A</b>      | save number 0-55 in E<br>for displacement                      |
| <b>MVI</b>    | <b>D,0</b>      | clear upper<br>displacement                                    |
| <b>LXI</b>    | <b>H,DGREEZ</b> | address of table to HL                                         |
| <b>DAD</b>    | <b>D</b>        | address + displacement<br>= address of required<br>table entry |
| <b>MOV</b>    | <b>A,M</b>      | get table entry                                                |
| <b>RSC</b>    |                 | clear carry                                                    |
| <b>RET</b>    |                 | return to calling<br>program                                   |
| <b>BUMMER</b> | <b>STC</b>      | carry on for error                                             |
|               |                 | return                                                         |
|               | <b>RET</b>      | return to calling<br>program                                   |
| <b>DGREEZ</b> | <b>DATA (—)</b> | temperature<br>corresponding to<br>voltage reading -20         |
|               | <b>DATA (—)</b> | temperature<br>corresponding to<br>voltage reading -19         |
|               | <b>etc.</b>     |                                                                |
|               | <b>...</b>      |                                                                |

Before leaving this example, note that an economy could have been made by combining the addition of the 20 with the test for the low limit. The sequence:

**CPI      -20**

```

J<    BUMMER
ADI    20

```

could be replaced with:

```

ADI    20
JM     BUMMER

```

which would save two words.

An even faster method for accomplishing this depends on a knowledge of the Z80's detailed binary instruction structure. The displacement of an indexed instruction is in the third word of the instruction. This displacement can be directly filled in by the program, thus saving the tedious address computation in DE and HL of example 10-7. The reader is warned that this method *cannot* be implemented in ROM, and further that it will be frowned upon by devotees of pure procedure programming and other priestly persons. The method is shown in example 10-8.

#### Example 10-8

Perform the table lookup described in example 10-7 by direct modification of the displacement of the indexed access instruction as described above.

| <i>label</i> | <i>inst.</i> | <i>operand</i> |                                                                   |
|--------------|--------------|----------------|-------------------------------------------------------------------|
| THRMS        | CPI          | 36             | test for upper limit                                              |
|              | J>=          | NOGOOD         |                                                                   |
|              | CPI          | -20            | test for lower limit                                              |
|              | J<           | NOGOOD         |                                                                   |
|              | STA          | INST+2         | store directly into displacement to LAX                           |
|              | LXI          | X,DGREEZ+20    | table address plus offset to X                                    |
| INST         | LAX          | 0              | fetch table item. The displacement of this LAX has been filled in |

**RSC**  
**RET**  
**NOGOOD STC**  
  
**RET**  
...

above with the incom-  
ing number, elimi-  
nating the address  
computation  
clear carry  
return  
set carry for error  
return  
return

While this is much shorter and faster than the conventional method, the reader is again warned that it will *not* work in ROM because of the direct modification of a program instruction.



## 11 / DECIMAL ARITHMETIC

“I hate definitions.”

Benjamin Disraeli, 1826

Before beginning to discuss decimal arithmetic it is necessary that we define exactly what is meant by a decimal number in a binary computer like the Z80. Decimal digits are represented as four bit groups which are the binary equivalents of the decimal digits. This system of representation is known as *binary coded decimal* or BCD. The decimal number 94 is represented in an eight bit Z80 word:

10010100

the upper four bits being the nine and the lower four bits the four. BCD constants can be generated by the assembler as a special case of hexadecimal, i.e.:

**ANGST DATA :94**

will generate the binary pattern shown above, while:

**DISMAL DBL :3147**

will generate:

*lower address*  
01000111

*higher address*  
00110001

In BCD the values  $1010_2$  through  $1111_2$  have no numerical meaning.

It must be understood at the outset that neither the Z80 nor any other existing microcomputer can perform decimal arithmetic. The

arithmetic of the Z80 is binary and only binary. What the Z80 can do is to rectify the result of a binary addition or subtraction of eight bits to yield a numerically correct BCD result. This is done by means of the **DAA** instruction. Consider the two BCD numbers:

and

|          |           |
|----------|-----------|
| 00100111 | $27_{10}$ |
| 00110101 | $35_{10}$ |

When these numbers are added with one of the various **ADD** instructions of the Z80 the result is:

01011100

which makes no sense in decimal. If the **DAA** instruction is executed immediately after the addition, the sum will be adjusted to:

01100010  $62_{10}$

the correct BCD result. Just how this is done is of no relevance here. The fact that it is done is all that matters for programming. The **DAA** must be executed after the addition and before any other instruction which alters the flags. The addition of a pair of BCD numbers is shown in example 11-1.

#### Example 11-1

A pair of two digit BCD numbers is in B and C. Form their sum and leave it in A.

| <i>label</i> | <i>inst.</i> | <i>operand</i> |                                       |
|--------------|--------------|----------------|---------------------------------------|
|              | <b>MOV</b>   | <b>A,B</b>     | augend to A                           |
|              | <b>ADD</b>   | <b>C</b>       | form sum with addend                  |
|              | <b>DAA</b>   |                | adjust for correct BCD representation |
|              | ...          |                |                                       |

The **DAA** will also set the carry bit if the BCD sum generated was greater than 99. This carry can be used to propagate the sum to higher orders, allowing multiple precision BCD addition. This is shown in example 11-2.

### Example 11-2

Double precision (4 digit) BCD numbers are in the BC and DE register pairs. Add the numbers and leave their sum in DE. Since addition of a pair of four digit numbers may result in a fifth digit, leave this digit in L.

| <i>label</i>  | <i>inst.</i> | <i>operand</i> |                                                                                     |
|---------------|--------------|----------------|-------------------------------------------------------------------------------------|
| <b>DBLADD</b> | <b>MOV</b>   | <b>A,C</b>     | low augend to A                                                                     |
|               | <b>ADD</b>   | <b>E</b>       | form low sum                                                                        |
|               | <b>DAA</b>   |                | adjust for correct BCD                                                              |
|               | <b>MOV</b>   | <b>E,A</b>     | save low sum                                                                        |
|               | <b>MOV</b>   | <b>A,B</b>     | get high augend                                                                     |
|               | <b>ADC</b>   | <b>D</b>       | form high sum                                                                       |
|               | <b>DAA</b>   |                | adjust for correct BCD                                                              |
|               | <b>MOV</b>   | <b>D,A</b>     | save high result                                                                    |
|               | <b>MVI</b>   | <b>A,0</b>     | <b>MVI A,0</b> is used<br>because it does not disturb the flags as <b>ZAR</b> would |
|               | <b>ACI</b>   | <b>0</b>       | propagate carry to fifth digit                                                      |
|               | <b>MOV</b>   | <b>L,A</b>     | fifth sum digit in L                                                                |
|               | <b>...</b>   |                |                                                                                     |

This process may be carried out to any required precision. The addition is carried out from low order to high, the lowest orders being added with a simple add, and all higher orders being added with carry from the lower orders.

It should be understood that the numbers being added *must* be correct BCD to begin with, or the result will not be valid. In particular, it must be clear that **DAA** *will not* perform binary to decimal

conversion. It will only rectify the result of the addition or subtraction of two proper BCD numbers. A long addition of the sort which might arise in a commercial application is shown in example 11-3.

### Example 11-3

A twelve digit BCD number at **TED** is to be added to a ten digit BCD number at **SINGER**, the sum being left in **TED**. Perform the addition using the X and Y registers as pointers.

| <i>label</i> | <i>inst.</i> | <i>operand</i>  |                                                                                                               |
|--------------|--------------|-----------------|---------------------------------------------------------------------------------------------------------------|
|              | <b>LXI</b>   | <b>X,TED</b>    | address of low order <b>TED</b> to X                                                                          |
|              | <b>LXI</b>   | <b>Y,SINGER</b> | address of low order <b>SINGER</b> to Y                                                                       |
|              | <b>LAX</b>   | <b>0</b>        | low order <b>TED</b> to A                                                                                     |
|              | <b>ADDY</b>  | <b>0</b>        | form low order sum                                                                                            |
|              | <b>DAA</b>   |                 | decimal adjust                                                                                                |
|              | <b>SAX</b>   | <b>0</b>        | store in low order <b>TED</b> .<br>Note that the low order addition was done with <b>ADDY</b> not <b>ADCY</b> |
|              | <b>LAX</b>   | <b>1</b>        | second order <b>TED</b> to A                                                                                  |
|              | <b>ADCY</b>  | <b>1</b>        | now <b>ADCY</b> is used to propagate the carry from the low order                                             |
|              | <b>DAA</b>   |                 | decimal adjust                                                                                                |
|              | <b>SAX</b>   | <b>1</b>        | store second order                                                                                            |
|              | <b>LAX</b>   | <b>2</b>        | get third order <b>TED</b> (5th and 6th digits)                                                               |
|              | <b>ADCY</b>  | <b>2</b>        | form third order sum                                                                                          |
|              | <b>DAA</b>   |                 | decimal adjust                                                                                                |
|              | <b>SAX</b>   | <b>2</b>        | store third order                                                                                             |
|              | <b>LAX</b>   | <b>3</b>        | get 7th and 8th digits of <b>TED</b>                                                                          |
|              | <b>ADCY</b>  | <b>3</b>        | form fourth order sum                                                                                         |

|             |          |                                  |
|-------------|----------|----------------------------------|
| <b>DAA</b>  |          | decimal adjust                   |
| <b>SAX</b>  | <b>3</b> | store fourth order               |
| <b>LAX</b>  | <b>4</b> | get 9th and 10th digits          |
|             |          | of <b>TED</b>                    |
| <b>ADCY</b> | <b>4</b> | form sum                         |
| <b>DAA</b>  |          | decimal adjust                   |
| <b>SAX</b>  | <b>4</b> | save fifth order. Now            |
|             |          | we have exhausted all            |
|             |          | the digits of <b>SINGER</b> ,    |
|             |          | but still have two digits        |
|             |          | of <b>TED</b> to go. This        |
|             |          | amounts only to propa-           |
|             |          | gating the carry from            |
|             |          | the fifth order addition         |
|             |          | into the sixth                   |
| <b>LAX</b>  | <b>5</b> | get 11th and 12th <b>TED</b>     |
|             |          | digits                           |
| <b>ACI</b>  | <b>0</b> | propagate carry                  |
| <b>DAA</b>  |          | decimal adjust highest           |
|             |          | order result                     |
| <b>SAX</b>  | <b>5</b> | and store in <b>TED</b> . If the |
|             |          | sum exceeded 12 deci-            |
|             |          | mal digits the carry bit         |
|             |          | would be on here                 |
|             |          | ...                              |

The **DAA** instruction can also rectify the result of a BCD subtraction. A simple single precision subtraction is shown in example 11-4.

#### Example 11-4

A pair of two digit BCD numbers is in the B and C registers. Find their BCD difference (B-C) and leave it in A.



| <i>label</i> | <i>inst.</i> | <i>operand</i> |                  |
|--------------|--------------|----------------|------------------|
|              | <b>MOV</b>   | <b>A,B</b>     | minuend to A     |
|              | <b>SUB</b>   | <b>C</b>       | minus subtrahend |
|              | <b>DAA</b>   |                | decimal adjust   |
|              |              |                | difference       |
|              | <b>...</b>   |                |                  |

As with addition, the carry bit is set in such a way as to allow propagation of a borrow to a higher order in multiple precision BCD subtraction. A double precision is shown in example 11-5.

#### Example 11-5

A pair of double precision BCD numbers is in BC and DE. Find their double precision difference (BC-DE) and leave it in DE.

| <i>label</i> | <i>inst.</i> | <i>operand</i> |                        |
|--------------|--------------|----------------|------------------------|
| <b>DSUB</b>  | <b>MOV</b>   | <b>A,C</b>     | low order minuend to A |
|              | <b>SUB</b>   | <b>E</b>       | form low binary        |
|              |              |                | difference             |
|              | <b>DAA</b>   |                | adjust to BCD          |
|              | <b>MOV</b>   | <b>E,A</b>     | save low order BCD     |
|              |              |                | difference             |
|              | <b>MOV</b>   | <b>A,B</b>     | high order minuend     |
|              |              |                | to A                   |
|              | <b>SBB</b>   | <b>D</b>       | form low BCD           |
|              |              |                | difference             |
|              | <b>DAA</b>   |                | adjust to BCD          |
|              | <b>MOV</b>   | <b>D,A</b>     | high order BCD         |
|              |              |                | difference to D        |
|              | <b>...</b>   |                |                        |

The absence of an algebraic sign in BCD arithmetic raises a problem when the subtrahend is greater than the minuend. In this case the

difference is represented as the ten's complement of the true BCD difference, corresponding to a negative result in signed arithmetic. This is always detectable by testing carry after the highest order subtraction and decimal adjust. If the result is valid BCD the carry bit will be off. If the result is a ten's complement negative number the carry bit will be on. What to do if this happens is a matter of the user's choice. One procedure would be to use one of the codes  $1010_2 - 1111_2$  as a sign, either beginning or ending each BCD string this way. Conversion of a ten's complement number to true BCD involves subtraction of each order from the maximum two digit BCD number, 99. The other alternative is to perform the subtraction again, swapping minuend and subtrahend.

There are two other Z80 instructions besides **DAA** which pertain to BCD arithmetic, **RLD** and **RRD**, meaning rotate right decimal and rotate left decimal. Both of these instructions operate between the low four bits of A and the memory word pointed to by HL. In **RRD** the low four bits of A replace the high four bits of the memory location and the high four bits of the memory word are moved to the low four bits of the memory word. The former low four bits of the memory word are loaded into the low four bits of A. The high four bits of A remain unchanged and do not participate in the operation. In **RLD** the high four bits of memory word are moved to the low four bits of A, the low four bits of the memory word move to the high four bits of the memory word and the former contents of the low four bits of A replace the low four bits of the memory word. Again, the high four bits of A remain unchanged and do not participate in the operation.

**RLD** and **RRD** allow true decimal shifts to be performed. A decimal left shift is shown in example 11-6.

#### Example 11-6

An eight digit BCD number is in memory at locations **NEWT** through **NEWT+3**, the lowest order being **NEWT**. Perform a decimal left shift of **NEWT**, ending with the digit shifted out in A.

| <i>label</i> | <i>inst.</i> | <i>operand</i> |                                                              |
|--------------|--------------|----------------|--------------------------------------------------------------|
| <b>DLSHF</b> | <b>LXI</b>   | <b>H,NEWT</b>  | address of NEWT<br>to HL                                     |
|              | <b>ZAR</b>   |                | clear A to shift in zero<br>at low end                       |
|              | <b>RLD</b>   |                | shift low order                                              |
|              | <b>INX</b>   | <b>H</b>       | pointer to second order                                      |
|              | <b>RLD</b>   |                | shift second order                                           |
|              | <b>INX</b>   | <b>H</b>       | pointer to third order                                       |
|              | <b>RLD</b>   |                | shift third order                                            |
|              | <b>INX</b>   | <b>H</b>       | pointer to highest order                                     |
|              | <b>RLD</b>   |                | shift highest order. The<br>digit shifted out is now<br>in A |
|              |              |                | ...                                                          |

In example 11-6 the A register was cleared before the first shift to bring a zero into the vacated digit position. If a digit were to be shifted into this position it should be loaded into the low order four bits of A before the first shift.

Decimal right shifts are done in essentially the same way except that the address in HL is that of the highest order and the pointer is decremented rather than incremented. This is obvious enough that an example here would be superfluous.

The ability to do true decimal shifts allows decimal multiplication and division to be performed fairly easily. This is a first for the Z80, the labor of shifting a BCD digit at a time having made the effort less than worthwhile on previous microcomputers.

Multiplication proceeds almost exactly as it does on paper. Multiplier digits are tested from right to left and the multiplicand added to a prezeroed product. At the end of each multiply step the multiplier is shifted left, corresponding to the indentation of the successive product lines in pencil and paper multiplication. The process is shown in example 11-7.

## Example 11-7

A subroutine is required to find the product of two four digit BCD numbers held in HL and DE. The high order product should be returned in HL and the low order in DE.

| <i>label</i>  | <i>inst.</i> | <i>operand</i>  |                                                                                                           |
|---------------|--------------|-----------------|-----------------------------------------------------------------------------------------------------------|
| <b>DCMPY</b>  | <b>SHLD</b>  | <b>MPLR</b>     | entry point, save multiplier                                                                              |
|               | <b>SDED</b>  | <b>MCND</b>     | save multiplicand                                                                                         |
|               | <b>LXI</b>   | <b>H,0</b>      | zero HL                                                                                                   |
|               | <b>SHLD</b>  | <b>MCND + 2</b> | clear four digits above multiplicand. This is because the multiplier will be shifted left into this space |
|               | <b>SHLD</b>  | <b>PROD</b>     | clear product area                                                                                        |
|               | <b>SHLD</b>  | <b>PROD + 2</b> | ...                                                                                                       |
|               | <b>LXI</b>   | <b>X,PROD</b>   | address of low order product to X                                                                         |
|               | <b>LXI</b>   | <b>Y,MCND</b>   | and low order multiplicand to Y                                                                           |
|               | <b>MVI</b>   | <b>B,4</b>      | multiply step count to B                                                                                  |
| <b>DCMPY1</b> | <b>PUSH</b>  | <b>B</b>        | save count on stack                                                                                       |
|               | <b>LXI</b>   | <b>H,MPLR+1</b> | address of high multiplier to HL                                                                          |
|               | <b>ZAR</b>   |                 | clear A to shift multiplier right                                                                         |
|               | <b>RRD</b>   |                 | shift high order multiplier right                                                                         |
|               | <b>DCX</b>   | <b>H</b>        | point to low order multiplier                                                                             |
|               | <b>RRD</b>   |                 | shift low order multiplier to right. The digit shifted out is in A                                        |
|               | <b>BZ</b>    | <b>DCMPY3</b>   | skip if multiplier digit zero                                                                             |

|               |                    |                                                                                                                                                                                       |
|---------------|--------------------|---------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------|
|               | <b>MOV B,A</b>     | otherwise use digit as counter                                                                                                                                                        |
| <b>DCMPY2</b> | <b>LAX 0</b>       | add multiplicand to product                                                                                                                                                           |
|               | <b>ADDY 0</b>      | ...                                                                                                                                                                                   |
|               | <b>DAA</b>         |                                                                                                                                                                                       |
|               | <b>SAX 0</b>       | save low order product                                                                                                                                                                |
|               | <b>LAX 1</b>       | second order                                                                                                                                                                          |
|               | <b>ADCY 1</b>      | <b>ADC</b> to propagate carry from low order                                                                                                                                          |
|               | <b>DAA</b>         |                                                                                                                                                                                       |
|               | <b>SAX 1</b>       | store second order product                                                                                                                                                            |
|               | <b>LAX 2</b>       | third order                                                                                                                                                                           |
|               | <b>ADCY 2</b>      | again, propagate carry                                                                                                                                                                |
|               | <b>DAA</b>         |                                                                                                                                                                                       |
|               | <b>SAX 2</b>       | save third order                                                                                                                                                                      |
|               | <b>LAX 3</b>       | fourth order                                                                                                                                                                          |
|               | <b>ADCY 3</b>      | ...                                                                                                                                                                                   |
|               | <b>DAA</b>         |                                                                                                                                                                                       |
|               | <b>SAX 3</b>       | save fourth order. The loop which begins at <b>DCMPY2</b> and ends with the <b>DBNZ</b> which follows is executed the number of times given by the multiplier digit shifted out above |
|               | <b>DBNZ DCMPY2</b> | repeat if multiplier digit not exhausted                                                                                                                                              |
| <b>DCMPY3</b> | <b>LXI H,MCND</b>  | now shift multiplicand left                                                                                                                                                           |
|               | <b>ZAR</b>         | clear A to avoid bringing in extraneous garbage into the low order digit                                                                                                              |



|             |               |                                |
|-------------|---------------|--------------------------------|
| <b>RLD</b>  |               | shift low order                |
| <b>INX</b>  | <b>H</b>      | point to second order          |
| <b>RLD</b>  |               | shift second order             |
| <b>INX</b>  | <b>H</b>      | point to third order           |
| <b>RLD</b>  |               | shift third order              |
| <b>INX</b>  | <b>H</b>      | point to fourth order          |
| <b>RLD</b>  |               | shift highest order            |
| <b>POP</b>  | <b>B</b>      | retrieve step count from stack |
| <b>DBNZ</b> | <b>DCMPY1</b> | repeat if count not zero       |
| <b>LDED</b> | <b>PROD</b>   | low order product to DE        |
| <b>LHLD</b> | <b>PROD+2</b> | high order to HL               |
| <b>RET</b>  |               |                                |
| <b>...</b>  |               |                                |

As the reader may have begun to guess, this is a very involved process which requires the execution of a great many more instructions than a binary multiplication. In addition, some of these instructions, particularly **RRD** and **RLD**, require considerable time to execute. The factors combine to make decimal multiplication almost glacially slow. If any significant amount of this must be done in a program it will pay the programmer to consider the alternative of converting the BCD to binary, performing the arithmetic in binary, then reconvert- ing the results to BCD.

Decimal division is also made feasible by the existence of the four bit decimal shifts. It is performed in a manner analogous to binary division, with a check for divide fault followed by the successive shifting left of the dividend and subtraction of the divisor from the high dividend. The subtraction is not a one-time affair as in binary division, however. It must be continued until the result becomes neg- ative, a prezeroed quotient digit being incremented for each success- ful subtraction. This agony is presented for your inspection in exam- ple 11-8.

## Example 11-8

An eight digit BCD dividend is in BC and HL, high order in BC, with a four digit BCD divisor in DE. Perform the indicated decimal division, returning the quotient in DE and the remainder in HL with carry off. If a divide fault occurs return with carry on.

| <i>label</i> | <i>inst.</i> | <i>operand</i>  |                                                                                                           |
|--------------|--------------|-----------------|-----------------------------------------------------------------------------------------------------------|
| <b>DCDVD</b> | <b>SHLD</b>  | <b>LDVDND</b>   | save low dividend                                                                                         |
|              | <b>SBCD</b>  | <b>HDVDND</b>   | and high dividend                                                                                         |
|              | <b>MOV</b>   | <b>A,C</b>      | perform divide fault                                                                                      |
|              |              |                 | test                                                                                                      |
|              | <b>SUB</b>   | <b>E</b>        | subtract low orders                                                                                       |
|              | <b>DAA</b>   |                 | decimal adjust                                                                                            |
|              | <b>MOV</b>   | <b>A,B</b>      | get high dividend                                                                                         |
|              | <b>SBB</b>   | <b>D</b>        | form high difference                                                                                      |
|              | <b>DAA</b>   |                 |                                                                                                           |
|              | <b>BNC</b>   | <b>DVDFLT</b>   | if carry is off the high dividend was greater than or equal to the divisor and the divide is not possible |
| <b>D2</b>    | <b>MVI</b>   | <b>B,4</b>      | divide step count to B                                                                                    |
|              | <b>LXI</b>   | <b>H,LDVDND</b> | low dividend address to HL                                                                                |
|              | <b>ZAR</b>   |                 | clear A for decimal shift                                                                                 |
|              | <b>MOV</b>   | <b>C,A</b>      | copy to C to clear quotient digit                                                                         |
|              | <b>RLD</b>   |                 | shift low order                                                                                           |
|              | <b>INX</b>   | <b>H</b>        | point to second order                                                                                     |
|              | <b>RLD</b>   |                 | shift second order                                                                                        |
|              | <b>INX</b>   | <b>H</b>        | point to third order                                                                                      |
|              | <b>RLD</b>   |                 | shift third order                                                                                         |
|              | <b>INX</b>   | <b>H</b>        | point to fourth order                                                                                     |

|    |             |               |                                                                                                                                  |
|----|-------------|---------------|----------------------------------------------------------------------------------------------------------------------------------|
| D3 | <b>RLD</b>  |               | shift fourth order                                                                                                               |
|    | <b>DCX</b>  | <b>H</b>      | move pointer back to beginning of high order dividend                                                                            |
|    | <b>MOV</b>  | <b>A,M</b>    | get lowest order high dividend                                                                                                   |
|    | <b>SUB</b>  | <b>E</b>      | subtract low divisor in E                                                                                                        |
|    | <b>DAA</b>  |               | decimal adjust                                                                                                                   |
|    | <b>PUSH</b> | <b>PSW</b>    | save on stack in case needed                                                                                                     |
|    | <b>INX</b>  | <b>H</b>      | point to highest order dividend                                                                                                  |
|    | <b>MOV</b>  | <b>A,M</b>    | fetch high two digits                                                                                                            |
|    | <b>SBB</b>  | <b>D</b>      | subtract high order                                                                                                              |
|    | <b>DAA</b>  |               | decimal adjust                                                                                                                   |
|    | <b>BC</b>   | <b>D4</b>     | if carry is on the difference was negative and the high divisor will remain unchanged, as if the subtraction had not taken place |
|    | <b>MOV</b>  | <b>M,A</b>    | replace high order dividend                                                                                                      |
|    | <b>DCX</b>  | <b>H</b>      | point to third order dividend                                                                                                    |
|    | <b>POP</b>  | <b>PSW</b>    | retrieve third order dividend from stack and replace third order                                                                 |
| D4 | <b>MOV</b>  | <b>M,A</b>    | increment quotient digit                                                                                                         |
|    | <b>INR</b>  | <b>C</b>      | and do it again until the subtraction gives a negative result                                                                    |
|    | <b>BRA</b>  | <b>D3</b>     |                                                                                                                                  |
|    | <b>POP</b>  | <b>PSW</b>    | restore stack pointer                                                                                                            |
|    | <b>LDA</b>  | <b>LDVDND</b> | get low dividend and merge                                                                                                       |
|    | <b>ORA</b>  | <b>C</b>      | new quotient digit into                                                                                                          |

|               |               |                                                              |
|---------------|---------------|--------------------------------------------------------------|
|               |               | the place vacated by the<br>decimal left shift above         |
| <b>STA</b>    | <b>LDVDND</b> |                                                              |
| <b>DBNZ</b>   | <b>D2</b>     | and do it again if divide<br>step count not yet<br>exhausted |
| <b>RSC</b>    |               | clear carry for normal<br>return                             |
| <b>LHLD</b>   | <b>HDVDND</b> | remainder to HL                                              |
| <b>LDED</b>   | <b>LDVDND</b> | quotient to DE                                               |
| <b>RET</b>    |               |                                                              |
| <b>DVDFLT</b> | <b>STC</b>    | set carry for divide fault                                   |
|               | <b>RET</b>    |                                                              |
|               | <b>...</b>    |                                                              |

Again, this process is very, very slow. Unless some compelling reason exists for staying with decimal, the reader is urged to consider conversion to binary for arithmetic of all kinds, with reconversion of the result for output. While the Z80's decimal arithmetic facilities are much better than those of any other existing microcomputer, they are still, in this writer's opinion, too primitive for serious use.

If the reader is determined to use decimal arithmetic, a valuable exercise would be to explore the case in which the decimal divide of example 11-8 fails, namely when the highest BCD dividend digit is nonzero. If this is thoroughly studied out you will have achieved a significant understanding of the process. A clue to the answer is provided in the earlier discussion of binary division. Good luck. Users of decimal arithmetic will need it, along with lots of patience and extra memory.

## 12 / COMMUNICATION WITH TERMINALS

“He was so fast he could turn the light out  
and be in bed before the room got dark.”

Description of “Cool Papa” Bell,  
baseball star of the old Negro Leagues

A computer program ultimately derives its meaning from actions taken by it in response to promptings from the real world, those actions, being communicated back to the real world in some form or other. The problem of coordinating the activity of the computer with that of the real world is made complicated by the great difference between the internal speed of the computer and the speed of the real world devices with which it communicates. The computer is very fast indeed — perhaps not as fast as “Cool Papa” Bell in the above quote — but very fast. Keyboards and printers, on the other hand, are geared more to the speeds with which humans are comfortable, and therein lies the problem of coordination of the input/output activity. It is the purpose of this chapter to detail how this coordination is achieved by programming. The chapter is by no means an exhaustive discussion of the input/output capabilities of the Z80, which would require a more extensive space. The information presented here will be enough to allow the beginning user to get started. Once these fundamentals have been mastered the reader can proceed to some of the more exotic data transfer modes available on the Z80.

A peripheral device is connected to the computer by a controller known as an *interface*. This interface may be simple, forcing the program to perform most of the coordination work, or it may be complex as in the case of a disk, in which data transfer speeds are such that the computer would have difficulty keeping up with data transfer while performing all of the subsidiary housekeeping tasks. In either case the interface and the peripheral device are indistin-



guishable to the program. All communication with the peripheral is done via the interface.

The most fundamental function of the peripheral interface is to distinguish signals from the computer which are intended for it from those intended for another peripheral. This is done by the assignment of a unique number or numbers to each device, this number or numbers being known as the *device address*. This is not to be confused with a memory address. Memory is not involved in the transfer of data in the Z80, at least not in the modes we will consider here.

There are three broad classes of operations performed by the computer in connection with input/output operations. They are:

- 1) Control functions. These involve a signal or signals sent from the computer to the peripheral interface to initiate some action. The functions vary from peripheral to peripheral.
- 2) Sense functions. These involve interrogation of the interface by the computer to determine the status of an operation in progress. The commonest of the sense functions is a test to see whether the peripheral is busy or ready to transmit data.
- 3) Data transfer. These operations move data between the computer and peripherals. Movement from the peripheral to the computer is known as input. Move of data from the computer to the peripheral is known as output.

Perhaps the commonest of all peripheral devices is a terminal of some kind, either a teletypewriter or one of the many CRT devices available, and it is to the servicing of these devices that we will first turn our attention. In many ways a terminal device is one of the more complex peripherals the reader is likely to experience, involving as it does both input from and output to the same physical device. If the principles of programming for one of these devices are fully understood the reader should have no trouble transferring the knowledge to less complex devices.

A terminal has two device addresses connected with it, usually but not necessarily adjacent numbers. These device addresses are known as the *control or status port* and the *data port*. Again, usually but not always, the control port has the lower address and the data port

the next higher address. This will vary from system to system. What will be discussed here is the system used by the writer in the development of examples for this book. The reader should clearly understand that while the overall approach to the problem is general, the details of the process, e.g., which bit in the control port does what, will very likely be different in the system he uses.

The writer's system has two terminal devices, a Teletype® connected to control port 0 and data port 1, and a CRT device connected to control port 20<sub>16</sub> and data port 21<sub>16</sub>. These devices are completely independent. We will begin with the Teletype. In the absence of any other Teletype activity, a character can be printed on the Teletype printer by:

|            |               |                            |
|------------|---------------|----------------------------|
| <b>MVI</b> | <b>A, '%'</b> | percent sign to A          |
| <b>OUT</b> | <b>1</b>      | send to Teletype data port |

This will print the percent sign (%) on the Teletype printer. This is simple enough if only the single character is to be printed. A complication arises if a string of characters is to be printed, however. Since the Teletype requires about  $1/10$  of a second to print each character sent to it, the computer must somehow arrange to pause between characters to allow the Teletype to process the character just sent before sending another out. There are a number of ways of doing this of which the simplest is to build a time delay into the program. A subroutine which does this is shown in example 12-1.

#### Example 12-1

A subroutine to print the character in A on entry is required. This subroutine should delay .1 second after the output and before returning to the calling program.

| <i>label</i> | <i>inst.</i> | <i>operand</i> |                                      |
|--------------|--------------|----------------|--------------------------------------|
| <b>ODLAY</b> | <b>OUT</b>   | <b>1</b>       | send character to Teletype data port |
|              | <b>LXI</b>   | <b>B,(---</b>  | a number is now loaded               |

|               |            |               |                                                                                                                                   |
|---------------|------------|---------------|-----------------------------------------------------------------------------------------------------------------------------------|
| <b>DLAYLP</b> | <b>DCX</b> | <b>B</b>      | into BC which will depend on the speed of the system. BC will be decremented to zero, the count being tested after each decrement |
|               | <b>MOV</b> | <b>A,B</b>    | decrement B                                                                                                                       |
|               | <b>ORA</b> | <b>C</b>      | now test for zero                                                                                                                 |
|               | <b>BNZ</b> | <b>DLAYLP</b> | both halves                                                                                                                       |
|               |            |               | decrement again if not yet zero                                                                                                   |
|               | <b>RET</b> |               |                                                                                                                                   |
|               | <b>...</b> |               |                                                                                                                                   |

This scheme will work. The constant operand of the **LXI** can be experimentally adjusted to make the device run at maximum speed. The A, B and C registers as well as the flags are destroyed by this procedure, however. If they are to be returned unmolested, as is often the requirement, the routine must save and restore them. The routine then looks like this:

| <i>label</i>  | <i>inst.</i> | <i>operand</i> |                                       |
|---------------|--------------|----------------|---------------------------------------|
| <b>ODLAY</b>  | <b>OUT</b>   | <b>1</b>       | send character to Teletype, as before |
|               | <b>PUSH</b>  | <b>PSW</b>     | save AF                               |
|               | <b>PUSH</b>  | <b>B</b>       | save BC                               |
|               | <b>LXI</b>   | <b>B,(---)</b> | count to BC                           |
| <b>DLAYLP</b> | <b>DCX</b>   | <b>B</b>       | count down, as before                 |
|               | <b>MOV</b>   | <b>A,B</b>     | ...                                   |
|               | <b>ORA</b>   | <b>C</b>       | ...                                   |
|               | <b>BNZ</b>   | <b>DLAYLP</b>  | ...                                   |
|               | <b>POP</b>   | <b>B</b>       | restore BC                            |
|               | <b>POP</b>   | <b>PSW</b>     | restore AF                            |
|               | <b>RET</b>   |                |                                       |
|               | <b>...</b>   |                |                                       |

If the system is sufficiently fast, the capacity of BC may not be sufficient to delay .1 second by merely counting. A more elaborate delay scheme must then be devised.

Using the subroutine of example 12-1 a string of characters may be printed by successive calls. The name LISA could be printed by:

```
MVI    A,'L'  
CALL   ODLAY  
MVI    A,'I'  
CALL   ODLAY  
MVI    A,'S'  
CALL   ODLAY  
MVI    A,'A'  
CALL   ODLAY  
...
```

This scheme would work, but it is clumsy, and would have to be reprogrammed if a terminal of different speed were substituted, e.g., one of the many 30 character per second devices now available.

A better way is to use the built in delay facilities of the interface to sense the terminal's readiness to accept another character. The busy/not busy status of the terminal is recorded in the bits of the control/status word. In the writer's system the leftmost bit of the control/status word is the busy/not busy indicator for Teletype output. This bit is a one if the a character is being printed and a zero if no activity is in progress, i.e., if the Teletype is ready to accept another character. The contents of the control/status port are loaded into the A register by:

```
IN      0
```

The use of the status bits in the control/status word for output is shown in example 12-2.

## Example 12-2

A subroutine is required to print the character in A on entry. Use the method just outlined. All registers should return unchanged.

| <i>label</i>  | <i>inst.</i> | <i>operand</i>  |                                                                                                      |
|---------------|--------------|-----------------|------------------------------------------------------------------------------------------------------|
| <b>TTYOUT</b> | <b>OUT</b>   | <b>TTYDTA</b>   | send character to TTY data port                                                                      |
|               | <b>PUSH</b>  | <b>PSW</b>      | save AF                                                                                              |
| <b>TTYOT2</b> | <b>IN</b>    | <b>TTYCTL</b>   | read control word                                                                                    |
|               | <b>RAL</b>   |                 | rotate leftmost bit into carry                                                                       |
|               | <b>BC</b>    | <b>TTYOT2</b>   | if carry is still a one the character is still being processed by the Teletype. Branch back and wait |
|               | <b>POP</b>   | <b>PSW</b>      | retrieve AF                                                                                          |
|               | <b>RET</b>   |                 | return to calling program                                                                            |
| <b>TTYCTL</b> | <b>EQU</b>   | <b>0</b>        | define TTY control/status port                                                                       |
| <b>TTYDTA</b> | <b>EQU</b>   | <b>TTYCTL+1</b> | define TTY data port                                                                                 |

This subroutine can be used to print a string of characters in the same way as was shown immediately following example 12-1. A more convenient way to print the string is shown in example 12-3.

## Example 12-3

Using the **TTYOUT** routine of example 12-2, write a subroutine to print a string of characters. The address of the first character to be printed is in HL and the number of characters in the string is in B.



| <i>label</i> | <i>inst.</i> | <i>operand</i> |                                     |
|--------------|--------------|----------------|-------------------------------------|
| <b>PSTRG</b> | <b>MOV</b>   | <b>A,M</b>     | get first character                 |
|              | <b>CALL</b>  | <b>TTYOUT</b>  | print it                            |
|              | <b>INX</b>   | <b>H</b>       | increment pointer                   |
|              | <b>DBNZ</b>  | <b>PSTRG</b>   | decrement count and repeat          |
|              | <b>RET</b>   |                | finished, return to calling program |
| <b>...</b>   |              |                |                                     |

Using this system the message:

### WHO DRANK MY BEER?

can be printed by:

|             |                       |                                         |
|-------------|-----------------------|-----------------------------------------|
| <b>LXI</b>  | <b>H,BRMSGE</b>       | address of <b>BEER</b><br>message to HL |
| <b>MVI</b>  | <b>B,LST+1-BRMSGE</b> | character count to B                    |
| <b>CALL</b> | <b>PSTRG</b>          | call the string print routine           |
| <b>...</b>  |                       |                                         |

where the message is defined elsewhere in the program by:

|               |             |                            |
|---------------|-------------|----------------------------|
| <b>BRMSGE</b> | <b>ASC</b>  | <b>'WHO DRANK MY BEER'</b> |
| <b>LST</b>    | <b>DATA</b> | <b>'?'</b>                 |

Very often it is inconvenient to specify a number of characters to be printed. Another scheme is to choose some special character as a message terminator. The string print routine prints characters until the agreed upon termination character is encountered in the string. Control is then returned to the calling program. The termination character itself is not printed. Such a string print routine is shown in example 12-4.

## Example 12-4

A string print subroutine is required which will accept the address of the first character to be printed in HL. The successive characters of the string are to be printed until a carriage return code ( $8D_{16}$ ) is encountered. At this point return control to the calling program. Do *not* output the carriage return.

| <i>label</i> | <i>inst.</i> | <i>operand</i> |                       |
|--------------|--------------|----------------|-----------------------|
| <b>PRT</b>   | <b>MOV</b>   | <b>A,M</b>     | get message character |
|              | <b>CPI</b>   | <b>:8D</b>     | check for carriage    |
|              |              |                | return                |
|              | <b>REQ</b>   |                | return if equal       |
|              | <b>CALL</b>  | <b>TTYOUT</b>  | otherwise print the   |
|              |              |                | character             |
|              | <b>INX</b>   | <b>H</b>       | increment the pointer |
|              | <b>BRA</b>   | <b>PRT</b>     | and do it again       |
|              |              |                | ...                   |

This form has the advantage that the number of characters to be printed need not be specified. The message is of the form:

|            |             |                                      |
|------------|-------------|--------------------------------------|
| <b>MSG</b> | <b>ASC</b>  | <b>'DARTH VADER IS A GOOD GUY!!'</b> |
|            | <b>DATA</b> | <b>:8D</b>                           |

and the routine is used by:

|             |              |                          |
|-------------|--------------|--------------------------|
| <b>LXI</b>  | <b>H,MSG</b> | address of first message |
|             |              | character to HL          |
| <b>CALL</b> | <b>PRT</b>   |                          |
|             |              | ...                      |

To emphasize the difference between different terminal devices, and to underscore that the input/output illustrations given here are

specific to the writer's system, the equivalent sequence to **TTYOUT** is given here for the CRT terminal attached to the same system.

|               |             |                 |                               |
|---------------|-------------|-----------------|-------------------------------|
| <b>CRTOUT</b> | <b>OUT</b>  | <b>CRTDTA</b>   | output to CRT data port       |
|               | <b>PUSH</b> | <b>PSW</b>      | save AF                       |
| <b>CRTOT2</b> | <b>IN</b>   | <b>CRTCTL</b>   | read control/status word      |
|               | <b>RAR</b>  |                 | move low bit to carry         |
|               | <b>RAR</b>  |                 | move next to low bit to carry |
|               | <b>BNC</b>  | <b>CRTOT2</b>   | skip if flag bit still zero   |
|               | <b>POP</b>  | <b>PSW</b>      | return to calling program     |
|               | <b>RET</b>  |                 |                               |
| <b>CRTDTA</b> | <b>EQU</b>  | <b>:20</b>      |                               |
| <b>CRTCTL</b> | <b>EQU</b>  | <b>CRTDTA+1</b> |                               |

Notice first that the busy flag for the CRT terminal is the second bit from the right, rather than the leftmost bit as for the Teletype. Notice also that the ready condition for the CRT is one, rather than the zero used with the Teletype. The reader must understand that these routines are purely illustrative and specific to a single system. They must be modified as indicated for use with different hardware.

Before leaving the subject of output to terminal devices a couple of things should be noted. First, the busy/not busy status of the output machinery is indicated by the zero or one status of a bit in the control/status word. The ready condition may be either zero or one depending on the specific interface device being used and system wiring. Second, the status of the output machinery will always be ready when it is not in use. This is of no importance now but will loom large when we come to the subject of output under interrupt control.

Input from a keyboard attached to a terminal is pretty much the inverse of output. The status of the input machinery is not ready until a key is struck. When the key is struck the flag indicates the ready state and the program may proceed to read the data port. This process is shown for Teletype in example 12-5.

## Example 12-5

The input ready bit is the rightmost bit of the Teletype control/status word. It is one in the not ready state and becomes zero when a key is struck. Write an instruction to read a character from the keyboard.

| <i>label</i> | <i>inst.</i> | <i>operand</i> |                         |
|--------------|--------------|----------------|-------------------------|
| <b>R1</b>    | <b>IN</b>    | <b>TTYCTL</b>  | read control word       |
|              | <b>RAR</b>   |                | rightmost bit to carry. |
|              |              |                | If this bit is a one no |
|              |              |                | key has been struck     |
|              | <b>BC</b>    | <b>R1</b>      | loop back until ready   |
|              | <b>IN</b>    | <b>TTYDTA</b>  | ready, read the         |
|              |              |                | character               |
|              |              |                | ...                     |

To again emphasize that each device is different, the input sequence for the CRT terminal in the writer's system is given below.

|             |            |               |                                           |
|-------------|------------|---------------|-------------------------------------------|
| <b>CRTR</b> | <b>IN</b>  | <b>CRTCTL</b> | read CRT control word                     |
|             | <b>RAR</b> |               | rightmost bit to carry.                   |
|             |            |               | This, by pure coincidence, is the same as |
|             |            |               | for the Teletype in example 12-5          |
|             | <b>BNC</b> | <b>CRTR</b>   | this time the test for                    |
|             |            |               | ready is for a one, not a                 |
|             |            |               | zero as for the Teletype                  |
|             | <b>IN</b>  | <b>CRTDTA</b> | ready, read the                           |
|             |            |               | character                                 |
|             |            |               | ...                                       |

A terminal keyboard resembles a typewriter keyboard in its superficial characteristics, but there are some important differences. The most important of these is that striking a key does *not* automatically

cause the character represented by that key to be printed on the printer or appear on the CRT screen. If this is to be done the program must supply the instructions to “echo” the character back to the terminal. Further, unlike a typewriter, a terminal device does *not* automatically advance one line when a carriage return code is sent out. The line advance, or line feed, is a separate function. Finally, the highest bit of a character received from the keyboard varies from device to device. It may be wired to be always a one or always a zero, or, it may be used to indicate the even or odd parity of the character as a check against errors in transmission. A blank may be represented by either  $00100000_2$  or  $10100000_2$ . To avoid having to check both cases some convention is adopted and the highest bit is forced to either zero or one. The high bit may be forced to zero by:

**ANI       :7F**

immediately after the character is read, or to one by:

**ORI       :80**

One or the other of these conventions must be used. The examples in this book and the software in the appendices assume that the highest bit is forced to one. A Teletype keyboard service subroutine which performs automatic line feeds and forces the high bit of the received character to one is shown in example 12-6.

#### Example 12-6

Write a Teletype keyboard service routine which forces the high bit of the typed character to one, and “echos” the typed character back to the Teletype printer. When a carriage return is received the line feed code ( $8A_{16}$ ) should be supplied automatically. The return in this last case should be with the carriage return code, not the line feed. The typed character is to be returned to the calling program in the A register.



| <i>label</i> | <i>inst.</i> | <i>operand</i> |                                             |
|--------------|--------------|----------------|---------------------------------------------|
| <b>RKB</b>   | <b>IN</b>    | <b>TTYCTL</b>  | get TTY control word                        |
|              | <b>RAR</b>   |                | low bit to carry                            |
|              | <b>BC</b>    | <b>RKB</b>     | loop until ready                            |
|              | <b>IN</b>    | <b>TTYDTA</b>  | ready read typed character                  |
| <b>RKB1</b>  | <b>ORI</b>   | <b>:80</b>     | force high bit to one                       |
|              | <b>PUSH</b>  | <b>PSW</b>     | save character                              |
|              | <b>OUT</b>   | <b>TTYDTA</b>  | send character to printer                   |
|              |              |                |                                             |
| <b>RKB2</b>  | <b>IN</b>    | <b>TTYCTL</b>  | get control word again                      |
|              | <b>RAL</b>   |                | leftmost bit to carry                       |
|              | <b>BC</b>    | <b>RKB2</b>    | loop until character clears                 |
|              |              |                |                                             |
|              | <b>POP</b>   | <b>PSW</b>     | restore AF                                  |
|              | <b>CPI</b>   | <b>:8D</b>     | was character just typed a carriage return? |
|              |              |                | return if not                               |
|              | <b>RNE</b>   |                |                                             |
|              | <b>MVI</b>   | <b>A,:8A</b>   | otherwise supply line feed code             |
|              |              |                |                                             |
|              | <b>OUT</b>   | <b>TTYDTA</b>  | send it out                                 |
|              | <b>IN</b>    | <b>TTYCTL</b>  | read status, as before                      |
| <b>RKB3</b>  | <b>RAL</b>   |                | ...                                         |
|              | <b>BC</b>    | <b>RKB</b>     | loop until line feed clears                 |
|              |              |                |                                             |
|              | <b>MVI</b>   | <b>A,:8D</b>   | reload carriage return code                 |
|              | <b>RET</b>   |                | and return                                  |
|              | <b>...</b>   |                |                                             |

The Teletype sequences given in this chapter are standard TTY sequences and device addresses for ALTAIR and IMSAI based machines. If the reader is using any other system the location of the ready bits and their one or zero status will, in general, be different. Consult the manufacturer's literature for the exact information for your system.

Another facet of terminal communication is the initialization of the terminal interface. There is no such thing as a standard initialization sequence. This sequence is necessary to set data rates and other parameters peculiar to each device. Again, consult the literature of the manufacturer of your system for specific details.

The Z80 is capable of performing input/output operations through registers other than A and in modes other than those already illustrated in this chapter. In particular, if the name of a single register is appended to the mnemonic like this:

### **INH**

a different sort of operation is performed. The device address in this case is presumed to be in the C register and the input/output operation is performed between this device address and the register named in the instruction mnemonic. Input is done by attaching any of the register names A, B, C, D, E, H, or L to the **IN** mnemonic. Output instructions are of the form:

### **OTA OTH**

etc. Again the device address is presumed to be in the C register, the data from the machine register (A or H in this case) going to that device address.

Even more elegant input/output modes are available using the **IN+**, **IN-**, **OUT+** and **OUT-** instructions. In this mode a count of the number of words to be transferred is kept in B and address of the first word of the block to be transferred in HL, with the device address in C. After each transfer the B count is decremented and HL is incremented or decremented, according to whether the mnemonic ended with + or -. The flags are set to show the condition of B after the decrementation.

Finally, there are the **IN+R**, **IN-R**, **OT+R** and **OT-R** instructions, which are the repeating counterparts of the instructions just discussed. They allow very high speed transfer of up to 256 words of data. This is useful when communicating with fast devices like disks.

# ASCII CODES

| <i>Character</i> | <i>Hexadecimal Code</i> | <i>Character</i> | <i>Hexadecimal Code</i> |
|------------------|-------------------------|------------------|-------------------------|
| @                | C0                      | blank            | A0                      |
| A                | C1                      | !                | A1                      |
| B                | C2                      | "                | A2                      |
| C                | C3                      | #                | A3                      |
| D                | C4                      | \$               | A4                      |
| E                | C5                      | %                | A5                      |
| F                | C6                      | &                | A6                      |
| G                | C7                      | '                | A7                      |
| H                | C8                      | (                | A8                      |
| I                | C9                      | )                | A9                      |
| J                | CA                      | *                | AA                      |
| K                | CB                      | +                | AB                      |
| L                | CC                      | ,                | AC                      |
| M                | CD                      | -                | AD                      |
| N                | CE                      | .                | AE                      |
| O                | CF                      | /                | AF                      |
| P                | D0                      | 0 (zero)         | B0                      |
| Q                | D1                      | 1                | B1                      |
| R                | D2                      | 2                | B2                      |
| S                | D3                      | 3                | B3                      |
| T                | D4                      | 4                | B4                      |
| U                | D5                      | 5                | B5                      |
| V                | D6                      | 6                | B6                      |
| W                | D7                      | 7                | B7                      |
| X                | D8                      | 8                | B8                      |
| Y                | D9                      | 9                | B9                      |
| Z                | DA                      | :                | BA                      |
| [                | DB                      | ;                | BB                      |
| \                | DC                      | <                | BC                      |
| ]                | DD                      | =                | BD                      |
| ↑                | DE                      | >                | BE                      |
| —                | DF                      | ?                | BF                      |

The following codes print nothing but perform the indicated functions.

|    |                 |    |        |
|----|-----------------|----|--------|
| 87 | ring bell       | FF | rubout |
| 8D | return carriage | 00 | null   |
| 8A | line feed       |    |        |

## NUMBER BASE CONVERSIONS

“Here is another bead on the string of confusions.”

*Meet General Grant*  
William E. Woodward

A computer engaged in any task other than pure digital meditation must eventually deal with the real world, taking data from it and reporting answers to it. Since the computer must deal with creatures who are captives of a ten fingered anatomy, the programmer is required to know how to convert between the internal binary system and the external real world system, usually decimal.

Systems for conversion between binary and decimal rely upon the fact that the coded representation of a decimal digit contains somewhere in it the equivalent binary quantity. Look for a moment at the ASCII codes for the decimal digits 0 - 9.

| <i>decimal digit</i> | <i>ASCII code</i> |
|----------------------|-------------------|
| 0                    | 1011 0000         |
| 1                    | 1011 0001         |
| 2                    | 1011 0010         |
| 3                    | 1011 0011         |
| 4                    | 1011 0100         |
| 5                    | 1011 0101         |
| 6                    | 1011 0110         |
| 7                    | 1011 0111         |
| 8                    | 1011 1000         |
| 9                    | 1011 1001         |

The ASCII codes have been deliberately split into four bit groups for easy examination. The upper four bits of the codes are identical. The lower four bits are simply the BCD representation of the digit. If the upper four bits can be gotten rid of somehow, the remaining



BCD will be easy to deal with. This can be done easily with the AND function. This operation and another important operation are shown in example 13-1.

### Example 13-1

A subroutine is required which will determine whether the character in A on entry is valid numeric, i.e., whether it lies in the  $B0_{16}$  through  $B9_{16}$  range. If the character is numeric mask out the upper four bits, returning with the resulting BCD in A and carry off. If the character is not valid numeric leave it unchanged and return with carry on.

| <i>label</i> | <i>inst.</i> | <i>operand</i> |                                                                                                                                                                                                                                                                  |
|--------------|--------------|----------------|------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------|
| <b>NCHK</b>  | <b>CPI</b>   | <b>:B0</b>     | check low limit                                                                                                                                                                                                                                                  |
|              | <b>BLS</b>   | <b>NON</b>     | non-numeric if less than $B0_{16}$                                                                                                                                                                                                                               |
|              | <b>CPI</b>   | <b>:BA</b>     | check high limit                                                                                                                                                                                                                                                 |
|              | <b>BGE</b>   | <b>NON</b>     | no good if greater than or equal to $BA_{16}$ . The test is made for one greater than the actual upper limit, $B9_{16}$ , because the Z80 has no test for greater than, only greater than or equal to. If control gets past here the character was valid numeric |
|              | <b>ANI</b>   | <b>:F</b>      | clear upper four bits and carry                                                                                                                                                                                                                                  |
|              | <b>RET</b>   |                | return to calling program                                                                                                                                                                                                                                        |
| <b>NON</b>   | <b>STC</b>   |                | carry on for error return                                                                                                                                                                                                                                        |
|              | <b>RET</b>   |                | return to calling program                                                                                                                                                                                                                                        |

The importance of the validity check in example 13-1 cannot be overstressed. The incoming data originate in some real world source which is subject to error, and if a finger slips while typing the consequences to the outcome of a program cannot be foreseen. *All* input, from any source, must be checked for validity and appropriateness. It is much better for the program to take a moment to ask the operator if he really means this than to have a mistake allow a process being controlled to run wild.

Now that we have a means of assurance of valid incoming data we can proceed with the conversion of the incoming number to binary. The simplest of all possible conversions is from decimal integer to binary integer. This is done by clearing a result area to zero and performing the following steps:

- a) get a digit and verify its validity
- b) multiply the previous result by ten
- c) add the BCD digit to the binary result
- d) start over with step a above

The multiplication by ten is simple, being done by the shift and add process shown in chapter 8. A single precision conversion of this type is shown in example 13-2.

#### Example 13-2

Using the **RKB** routine of chapter 12 and the validity check routine of example 13-1, create a subroutine which will convert a decimal number typed at the keyboard. The result should be returned in B. Conversion should cease when a non-numeric character is typed, and control returned to the calling program with the character which caused termination in A.

| <i>label</i> | <i>inst.</i> | <i>operand</i> |                                |
|--------------|--------------|----------------|--------------------------------|
| SCON         | MVI          | B,0            | clear result register          |
| SC1          | CALL         | RKB            | get character from<br>keyboard |

|             |             |                           |
|-------------|-------------|---------------------------|
| <b>CALL</b> | <b>NCHK</b> | check it for validity     |
| <b>RC</b>   |             | <b>NCHK</b> returns with  |
|             |             | carry on if the character |
|             |             | was non-numeric           |
| <b>MOV</b>  | <b>C,A</b>  | save typed, stripped      |
|             |             | BCD digit                 |
| <b>MOV</b>  | <b>A,B</b>  | get previous result       |
| <b>LLA</b>  |             | shift left                |
| <b>MOV</b>  | <b>B,A</b>  | save                      |
| <b>LLA</b>  |             | shift left again          |
| <b>LLA</b>  |             | and again                 |
| <b>ADD</b>  | <b>B</b>    | former result times 10    |
| <b>ADD</b>  | <b>C</b>    | add new digit             |
| <b>MOV</b>  | <b>B,A</b>  | save new result           |
| <b>BRA</b>  | <b>SC1</b>  | and get another digit     |
| <b>...</b>  |             |                           |

The program in example 13-2 will convert successive typed digits to a binary number — if there are not too many of them. The problem is that there is no check for the overflow of the result, i.e., the decimal number 256 would return as zero, the only significant bit being the ninth bit — which was shifted out. The remedy for this would be to check carry after each shift. If carry were a one after any shift the result overflowed.

An eight bit convert is not a very useful program, however. The extreme shortness of the Z80 word does us no good here. A practical converter would need to handle 16 bit numbers at least. Such a converter, with check for overflow, is shown in example 13-3.

### Example 13-3

A subroutine is required to convert a series of typed digits to a 16 bit binary integer. The calling sequence for the subroutine is to look like this:

| <i>label</i> | <i>inst.</i> | <i>operand</i> |                                     |
|--------------|--------------|----------------|-------------------------------------|
|              | <b>CALL</b>  | <b>CNV16</b>   | call converter                      |
|              | <b>BRA</b>   | <b>DSASTR</b>  | return here if result<br>overflowed |
|              | <b>...</b>   |                | return here if result<br>valid      |

Return is to be made with the binary result in HL and the character which caused termination in A.

|               |             |              |                                                                                                         |
|---------------|-------------|--------------|---------------------------------------------------------------------------------------------------------|
| <b>CNV16</b>  | <b>LXI</b>  | <b>H,0</b>   | clear result area                                                                                       |
|               | <b>MOV</b>  | <b>B,H</b>   | and upper addend for<br>new digit                                                                       |
| <b>CNV16A</b> | <b>CALL</b> | <b>RKB</b>   | get character from<br>keyboard                                                                          |
|               | <b>CALL</b> | <b>NCHK</b>  | check for numeric<br>validity                                                                           |
|               | <b>BC</b>   | <b>EOCNV</b> | skip if not numeric                                                                                     |
|               | <b>DAD</b>  | <b>H</b>     | shift HL left                                                                                           |
|               | <b>RC</b>   |              | check for overflow                                                                                      |
|               | <b>MOV</b>  | <b>D,H</b>   | save shifted number                                                                                     |
|               | <b>MOV</b>  | <b>E,L</b>   | ...                                                                                                     |
|               | <b>DAD</b>  | <b>H</b>     | shift again                                                                                             |
|               | <b>RC</b>   |              | check for overflow                                                                                      |
|               | <b>DAD</b>  | <b>H</b>     | shift again                                                                                             |
|               | <b>RC</b>   |              | check for overflow                                                                                      |
|               | <b>DAD</b>  | <b>D</b>     | add twice the original<br>number                                                                        |
|               | <b>RC</b>   |              | check again for<br>overflow                                                                             |
|               | <b>MOV</b>  | <b>C,A</b>   | move new digit to C.<br>BC will be added to<br>HL. Note that B has<br>already been set to zero<br>above |
|               | <b>DAD</b>  | <b>B</b>     | add new BCD digit                                                                                       |
|               | <b>RC</b>   |              | check for overflow                                                                                      |

|              |             |               |                         |
|--------------|-------------|---------------|-------------------------|
| <b>EOCNV</b> | <b>BRA</b>  | <b>CNV16A</b> | and do it again         |
|              | <b>XTHL</b> |               | swap result with return |
|              | <b>INX</b>  | <b>H</b>      | pointer at top of stack |
|              | <b>INX</b>  | <b>H</b>      | bump return pointer     |
|              | <b>XTHL</b> |               | twice                   |
|              | <b>RET</b>  |               | to skip over BRA        |
|              |             |               | instruction             |
|              |             |               | reswap                  |
|              |             |               | and return to calling   |
|              |             |               | program                 |
|              |             |               |                         |
|              |             |               |                         |

In some cases 16 bits will not be enough and it will be necessary to go to even greater precision. This operation involves the same principles as shown for single and double precision. The exact mechanics of such a conversion becomes a little more involved and tedious, as one might expect. A quadruple precision decimal to binary conversion is shown in example 13-4. This will convert a number as large as 4,294,967,303. A need for greater precision than this is seldom seen.

#### Example 13-4

A quadruple precision decimal to binary convert subroutine is required which will accept input from the keyboard and return the high order result in DE, the low order in HL. Normal return should skip the two word instruction following the **CALL**. If overflow occurs return to the instruction immediately following the **CALL**.

|              |              |                |                     |
|--------------|--------------|----------------|---------------------|
| <i>label</i> | <i>inst.</i> | <i>operand</i> |                     |
| <b>QCON</b>  | <b>LXI</b>   | <b>H,0</b>     | clear result area   |
|              | <b>MOV</b>   | <b>D,H</b>     | ...                 |
|              | <b>MOV</b>   | <b>E,H</b>     | ...                 |
| <b>QC1</b>   | <b>CALL</b>  | <b>RKB</b>     | get typed character |



|             |              |                                                                                                                                                                                                                                                                                                                                    |
|-------------|--------------|------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------|
| <b>CALL</b> | <b>NCHK</b>  | check validity                                                                                                                                                                                                                                                                                                                     |
| <b>BC</b>   | <b>EOQC</b>  | skip out if not numeric                                                                                                                                                                                                                                                                                                            |
| <b>CALL</b> | <b>DLSHF</b> | the shifting is relegated to a subroutine because it will be done 3 times                                                                                                                                                                                                                                                          |
|             |              | save intermediate result                                                                                                                                                                                                                                                                                                           |
|             |              | ...                                                                                                                                                                                                                                                                                                                                |
| <b>SDED</b> | <b>HI</b>    |                                                                                                                                                                                                                                                                                                                                    |
| <b>SHLD</b> | <b>LO</b>    |                                                                                                                                                                                                                                                                                                                                    |
| <b>CALL</b> | <b>DLSHF</b> | shift again                                                                                                                                                                                                                                                                                                                        |
| <b>CALL</b> | <b>DLSHF</b> | twice                                                                                                                                                                                                                                                                                                                              |
| <b>LBCD</b> | <b>LO</b>    | now add saved quantity                                                                                                                                                                                                                                                                                                             |
| <b>DAD</b>  | <b>B</b>     | ...                                                                                                                                                                                                                                                                                                                                |
| <b>XCHG</b> |              |                                                                                                                                                                                                                                                                                                                                    |
| <b>LBCD</b> | <b>HI</b>    | high order add                                                                                                                                                                                                                                                                                                                     |
| <b>DAC</b>  | <b>B</b>     | add high orders, propagate carry                                                                                                                                                                                                                                                                                                   |
|             |              | if there was a carry here the result is too big to be held                                                                                                                                                                                                                                                                         |
| <b>RC</b>   |              |                                                                                                                                                                                                                                                                                                                                    |
|             |              |                                                                                                                                                                                                                                                                                                                                    |
| <b>XCHG</b> |              | low order to HL again                                                                                                                                                                                                                                                                                                              |
| <b>MOV</b>  | <b>C,A</b>   | new digit to C                                                                                                                                                                                                                                                                                                                     |
| <b>MVI</b>  | <b>B,0</b>   | clear B for double add                                                                                                                                                                                                                                                                                                             |
| <b>DAD</b>  | <b>B</b>     | add new digit, low order                                                                                                                                                                                                                                                                                                           |
|             |              |                                                                                                                                                                                                                                                                                                                                    |
| <b>BNC</b>  | <b>QC1</b>   | if there is no carry here we can proceed with the next digit, otherwise increment high order. Here we have the problem again that <b>INR</b> does not set carry. It does set the zero bit, however, and the only condition under which a carry would be generated by incrementing would be if the original number were $FF_{16}$ , |
| <b>INR</b>  | <b>E</b>     |                                                                                                                                                                                                                                                                                                                                    |

```

      BNZ    QC1
      INR    D
      BNZ    QC1
      RET

```

```

DLSHF DAD    H
      XCHG
      DAC    H

      XCHG
      RNC

```

```

      INX    SP

```

```

      INX    SP

```

```

      RET

```

which is incremented to 00. Testing the zero flag will therefore amount to the same thing as testing carry  
 skip if no carry  
 increment highest order  
 skip back if no carry  
 if it gets here the converted number is too big to be held. Return to location immediately after **CALL**  
 shift low order left  
 swap for high order  
 shift high order, propagate carry  
 reswap  
 return if no carry. If there is a carry here the number overflowed 32 bits  
 the return to the calling program is a little complicated by the fact that the last item pushed onto the stack was not the return address for **QCON** but the return address for **DLSHF**. This is remedied by simply incrementing **SP** over the **DLSHF** return address to the **QCON** return address  
 take error return

|             |             |          |                                     |
|-------------|-------------|----------|-------------------------------------|
| <b>EOQC</b> | <b>XTHL</b> |          | swap HL with top of stack           |
|             | <b>INX</b>  | <b>H</b> | increment return address over error |
|             | <b>INX</b>  | <b>H</b> | return                              |
|             | <b>XTHL</b> |          | reswap                              |
|             | <b>RET</b>  |          | take normal return                  |
|             | ...         |          |                                     |

Conversion of decimal fractions to binary fractions begins by converting the successive digits as an integer, then dividing by ten a number of times equal to the number of decimal digits which appeared to the right of the point. This division is performed by using the highest order of the preliminary integer result as the low order dividend and zero as the high order dividend. With ten as the divisor this is guaranteed to be divisible. The quotient of this division is the high order quotient. The remainder forms the high order divisor for the next stage division, the low order divisor being the next order of the preliminary integer result. This is repeated until all of the integer orders have been divided, the successive quotients replacing the integer result. This entire process is repeated as many times as there were digits to the right of the point.

Note before leaving this fraction question that there are decimal fractions which cannot be exactly represented in binary, just as some fractions cannot be represented in decimal, e.g.,  $1/3$ ,  $1/7$ ,  $1/9$  etc. In particular the decimal fraction 0.1 is represented in binary by:

0.000110011001100110011001100 . . .

i.e., it is a repeating fraction. If this is converted to decimal the result is .099984741, approximately.

Conversion of binary integers to decimal is also a fairly straightforward process. There are two principal methods that can be used. The first is to divide the integer by ten, each successive division yielding a remainder. The first remainder is the BCD low order digit. The quotient from this division is then again divided by ten. This remain-

der is the BCD tens digit. The quotient at each stage is divided by ten to yield the next higher order BCD digit as a remainder. This process continues until the quotient is zero. A conversion of this type is shown in example 13-5.

### Example 13-5

Using the division scheme of example 8-9 and the Teletype output subroutine of example 12-2, create a subroutine to convert the unsigned binary integer in A to decimal. Print the decimal result on the Teletype. The program in example 8-9 is made into a subroutine by simply adding an **RET** to the end. It now looks like this:

| <i>label</i>  | <i>inst.</i> | <i>operand</i> |                                  |
|---------------|--------------|----------------|----------------------------------|
| <b>DVD</b>    | <b>MOV</b>   | <b>A,H</b>     | commentary as for<br>example 8-9 |
|               | <b>ORA</b>   | <b>A</b>       |                                  |
|               | <b>JM</b>    | <b>DVDFLT</b>  |                                  |
|               | <b>CMP</b>   | <b>D</b>       |                                  |
|               | <b>BGE</b>   | <b>DVDFLT</b>  |                                  |
|               | <b>MVI</b>   | <b>B,8</b>     |                                  |
|               | <b>MVI</b>   | <b>E,0</b>     |                                  |
| <b>DVD1</b>   | <b>DAD</b>   | <b>H</b>       |                                  |
|               | <b>DSC</b>   | <b>D</b>       |                                  |
|               | <b>BC</b>    | <b>DVD2</b>    |                                  |
|               | <b>INX</b>   | <b>H</b>       |                                  |
|               | <b>BRA</b>   | <b>DVD3</b>    |                                  |
| <b>DVD2</b>   | <b>DAD</b>   | <b>D</b>       |                                  |
| <b>DVD3</b>   | <b>DBNZ</b>  | <b>DVD1</b>    |                                  |
|               | <b>RSC</b>   |                | carry off for normal<br>return   |
|               | <b>RET</b>   |                |                                  |
| <b>DVDFLT</b> | <b>STC</b>   |                | carry on for divide fault        |
|               | <b>RET</b>   |                | return                           |

The sharp eyed reader may notice as he works through

the example which follows that it was not necessary to include the divide fault check, since no divide fault condition can arise, by the conditions of the problem. To create a divide routine without this check would sooner or later lead to a catastrophe when the routine was used by another program which did not guarantee against divide fault.

The conversion looks like this

|             |             |                                                                                                                                                                                                                                                                      |
|-------------|-------------|----------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------|
| <b>MOV</b>  | <b>L,A</b>  | move number to be converted to L                                                                                                                                                                                                                                     |
| <b>MVI</b>  | <b>H,0</b>  | clear upper dividend                                                                                                                                                                                                                                                 |
| <b>MVI</b>  | <b>D,10</b> | set divisor to 10. Since the high dividend is zero and the divisor 10 no divide fault can occur                                                                                                                                                                      |
| <b>CALL</b> | <b>DVD</b>  | perform the division. The remainder from this division in H is the low order digit. We can't print this now since printing goes from left to right and the two higher order digits have to be printed first. The easiest way to save it is to push it onto the stack |
| <b>PUSH</b> | <b>H</b>    | save low order (units) digit                                                                                                                                                                                                                                         |
| <b>MVI</b>  | <b>H,0</b>  | again clear high dividend. The quotient from the first division, still in L, will be the low dividend for the next division                                                                                                                                          |



**CALL DVD**

divide again. The remainder in H is the tens digit. The quotient in L is the hundreds digit since an eight bit number cannot have a value greater than  $255_{10}$

**MOV A,L**

hundreds digit to A  
form ASCII

**ORI :B0**

**CALL TTYOUT**

print hundreds digit

**MOV A,H**

tens digit to A

**ORI :B0**

form ASCII

**CALL TTYOUT**

print tens digit

**POP PSW**

recall that the units digit was saved by pushing HL onto the stack.

The BCD digit was in H, the higher order register. Popping the former HL contents into AF puts the former contents of H into A (Remember A is the high order register of the AF pair)

**ORI :B0**

form ASCII

**CALL TTYOUT**

print units digit

...

The essential awkwardness of the method of example 13-5 is that the digits are produced in the opposite order from that in which they are required for printing. This could be solved by pushing each generated digit in a long conversion onto the stack, allowing them to be retrieved in the opposite order from that in which they were pushed, i.e., the correct order for printing. Still, this wastes stack space and is clumsy. There is a better way which will produce the digits in the

order required for printing. This method involves starting with the highest possible decimal power, subtracting that power from the number to be converted until the result is negative. The number of successful subtractions, i.e. subtractions which did not have a negative result, is the high BCD digit. This power of ten is then added back into the number to give the value before the terminating subtraction. The next lower power of ten is then successively subtracted from the remaining number until it goes negative. The number of successful subtractions is the BCD digit at that decimal position. The power of ten is then added back in to make the number positive and the entire process is repeated until the units digit is reached. This sounds more tedious than it really is. The process is shown in example 13-6.

#### Example 13-6

Write an instruction sequence which will convert the unsigned eight bit number in A by the subtraction method just described and print it using the **TTYOUT** subroutine.

| <i>label</i> | <i>inst.</i> | <i>operand</i> |                                                                         |
|--------------|--------------|----------------|-------------------------------------------------------------------------|
|              | <b>MVI</b>   | <b>B,:B0</b>   | ASCII zero to B. This could as well have been written: <b>MVI B,'0'</b> |
| <b>C1</b>    | <b>SUI</b>   | <b>100</b>     | subtract 100's till it goes negative                                    |
|              | <b>BC</b>    | <b>C2</b>      | carry on means zero was crossed                                         |
|              | <b>INR</b>   | <b>B</b>       | increment ASCII digit in B                                              |
|              | <b>BRA</b>   | <b>C1</b>      | and subtract again                                                      |
| <b>C2</b>    | <b>ADI</b>   | <b>100</b>     | add the last 100 back in                                                |
|              | <b>PUSH</b>  | <b>PSW</b>     | save number                                                             |
|              | <b>MOV</b>   | <b>A,B</b>     | get generated ASCII digit                                               |
|              | <b>CALL</b>  | <b>TTYOUT</b>  | print high order digit                                                  |
|              | <b>MVI</b>   | <b>B,:B0</b>   | ASCII zero to B                                                         |

|           |             |               |                                   |
|-----------|-------------|---------------|-----------------------------------|
|           | <b>POP</b>  | <b>PSW</b>    | retrieve number from stack        |
| <b>C3</b> | <b>SUI</b>  | <b>10</b>     | subtract 10 till it goes negative |
|           | <b>BC</b>   | <b>C4</b>     | ...                               |
|           | <b>INR</b>  | <b>B</b>      | increment ASCII digit as before   |
|           | <b>BRA</b>  | <b>C3</b>     | ...                               |
| <b>C4</b> | <b>ADI</b>  | <b>10</b>     | add the last 10 back in           |
|           | <b>PUSH</b> | <b>PSW</b>    | save remaining BCD units digit    |
|           | <b>MOV</b>  | <b>A,B</b>    | get tens digit from B             |
|           | <b>CALL</b> | <b>TTYOUT</b> | print it                          |
|           | <b>POP</b>  | <b>PSW</b>    | retrieve BCD units digit          |
|           | <b>ORI</b>  | <b>:B0</b>    | merge with ASCII code bits        |
|           | <b>CALL</b> | <b>TTYOUT</b> | print units digit                 |
|           | ...         |               |                                   |

The more experienced programmer will notice an economy that could have been made here. The line:

```
C4      ADI      10
```

could have been replaced with:

```
C4      ADI      :BA
```

which would have allowed the removal of the **ORI :B0** in the second to the last line. Think about it.

Double precision and signed conversions involve the same methods as have been shown. A requirement which often arises is elimination of leading nonsignificant zero digits, the leading zeros being replaced by blank or some other character, i.e., the converted number should print like this:

instead of:

00485

In commercial applications particularly, numbers representing money often have leading zeros replaced with asterisks so that the amount cannot be changed, like this:

\*\*\*343.87

A scheme for doing this is shown in example 13-7.

#### Example 13-7

A subroutine is required which will convert the 16 bit unsigned magnitude in HL to a five digit decimal ASCII number, leaving this result in the block of five memory locations beginning at the address pointed to by X. Leading nonsignificant zeros should be replaced by blanks.

| <i>label</i> | <i>inst.</i> | <i>operand</i> |                                     |
|--------------|--------------|----------------|-------------------------------------|
| DCNV         | MVI          | C, ' '         | ASCII blank to C                    |
|              | MVI          | B, 5           | decimal digit count to B            |
|              | LXI          | Y, PTBL        | address of powers of ten table to Y |
| DCNV1        | LEY          | 0              | low order power of ten to E         |
|              | INX          | Y              | increment power table pointer       |
|              | LDY          | 0              | high order power of ten to D        |
|              | INX          | Y              | increment for next pass             |
|              | ZAR          |                | clear A for decimal result          |
| DCNV2        | DSC          | D              | subtract power of ten from number   |
|              | BC           | DCNV3          | carry if zero was                   |

**INR**     **A**  
**BRA**     **DCNV2**  
**DCNV3 DAD**     **D**  
**ORA**     **A**  
**BZ**     **DCNV4**  
**MVI**     **C,'0'**

**DCNV4 ORA**     **C**

**SAX**     **0**  
**INX**     **X**  
**DBNZ**     **DCNV1**

crossed  
 increment BCD digit  
 and do it again  
 restore number  
 was the digit zero?  
 ...  
 no, replace the filler  
 character with an  
 ASCII zero  
 if any nonzero BCD  
 digit has been generat-  
 ed, the ASCII blank in  
 C has been replaced by  
 ASCII zero. If the  
 blank is still there, it  
 will be ORed with zero  
 to give a blank. All  
 leading zero digits are  
 made blank by this  
 method  
 store character in string  
 increment store pointer  
 decrement count and  
 try again. When the  
 count goes to zero the  
 conversion is finished  
 except for one small  
 detail. If the original  
 number was all zero,  
 there were four leading  
 nonsignificant zeros  
 converted, i.e. no nu-  
 meric digits have been  
 stored, only blanks.  
 There must be at least  
 one numeric digit  
 displayed. We insure



**LAX     -1**

**ORI     :B0**

**SAX     -1**

**RET**

that this is so by:  
retrieve last character.  
Since X was already in-  
cremented before the  
**DBNZ** was executed,  
we  
use a displacement of  
-1 here to retrieve it  
force the last character  
numeric  
and put it back in the  
stored string

And the table of powers of ten:

|             |            |              |
|-------------|------------|--------------|
| <b>PTBL</b> | <b>DBL</b> | <b>10000</b> |
|             | <b>DBL</b> | <b>1000</b>  |
|             | <b>DBL</b> | <b>100</b>   |
|             | <b>DBL</b> | <b>10</b>    |
|             | <b>DBL</b> | <b>1</b>     |

The program of example 13-7 could have been made somewhat faster by making a special case of the units digit. After the last ten has been subtracted from the number the remaining number is, by definition, less than ten, i.e., it *is* the BCD units digit. Counting down again by subtracting one and adding one to the counter in A wastes time but avoids making a special case of the units digit, saving program space. Since conversion is done as part of the output process, the speed at which things happen will be determined by the speed of the peripheral receiving the information, hundreds of times slower than the rate at which the computer can generate it, so it was judged here better to save space rather than time.

Conversion of signed numbers is a little more difficult. Note is made of the algebraic sign of the number, and then it is forced positive, the conversion taking place on the positive number and the ap-

appropriate ASCII sign being attached. This process is shown in example 13-8. Note that this subroutine makes no special provision for the maximum negative number, which cannot be two's complemented, yet it converts this number correctly. Example 13-8 should be studied until you understand why this takes place.

### Example 13-8

A signed 16 bit number is in HL. A subroutine is required which will convert it to ASCII decimal, leaving the result in the six word block pointed to by X. The algebraic sign should be placed immediately to the left of the highest significant digit, and all leading nonsignificant zeros should be replaced by blanks.

| <i>label</i> | <i>inst.</i> | <i>operand</i> |                                                                          |
|--------------|--------------|----------------|--------------------------------------------------------------------------|
| SDCNV        | MVI          | B,'+'          | ASCII plus sign to B                                                     |
|              | ZAR          |                | clear A to determine sign                                                |
|              | ORA          | H              | high order binary to A                                                   |
|              | JP           | SDCNV2         | skip if positive                                                         |
|              | MVI          | B,'-'          | ASCII minus sign to B                                                    |
|              | MOV          | A,H            | now two's complement the number                                          |
|              | CMA          |                | ...                                                                      |
|              | MOV          | H,A            | ...                                                                      |
|              | MOV          | A,L            | ...                                                                      |
|              | CMA          |                | ...                                                                      |
|              | MOV          | L,A            | ...                                                                      |
|              | INX          | H              | if overflow occurs here the negation was illegal, but we ignore it. Why? |
| SDCNV2       | LXI          | Y,PTBL         | address of powers of ten table to Y                                      |
|              | INX          | X              | this makes the character store pointer point to the second of the six    |

|               |            |               |                                                                                                                                            |
|---------------|------------|---------------|--------------------------------------------------------------------------------------------------------------------------------------------|
|               | <b>MOV</b> | <b>A,B</b>    | words. There can be only five numeric digits.                                                                                              |
|               | <b>XAF</b> |               | The sixth space must hold either a blank or the algebraic sign                                                                             |
|               | <b>MVI</b> | <b>B,5</b>    | ASCII sign to A                                                                                                                            |
|               | <b>MVI</b> | <b>C,' '</b>  | save it in A'                                                                                                                              |
|               | <b>SCX</b> | <b>-1</b>     | digit count to B                                                                                                                           |
|               |            |               | ASCII blank to C                                                                                                                           |
|               |            |               | this stores the blank in C into the first word of the string. Since X had already been incremented the negative displacement was necessary |
| <b>SDCNV3</b> | <b>LEY</b> | <b>0</b>      | low order power of ten to E                                                                                                                |
|               | <b>INX</b> | <b>Y</b>      | increment pointer                                                                                                                          |
|               | <b>LDY</b> | <b>0</b>      | high order power of ten to D                                                                                                               |
|               | <b>INX</b> | <b>Y</b>      | increment pointer                                                                                                                          |
|               | <b>ZAR</b> |               | clear A to receive result digit                                                                                                            |
| <b>SDCNV4</b> | <b>DSC</b> | <b>D</b>      | subtract power of ten                                                                                                                      |
|               | <b>JM</b>  | <b>SDCNV5</b> | skip if it went negative                                                                                                                   |
|               | <b>INR</b> | <b>A</b>      | otherwise increment digit in A                                                                                                             |
|               | <b>BRA</b> | <b>SDCNV4</b> | and do it again                                                                                                                            |
| <b>SDCNV5</b> | <b>DAD</b> | <b>D</b>      | restore number                                                                                                                             |
|               | <b>ORA</b> | <b>A</b>      | test digit for zero                                                                                                                        |
|               | <b>BZ</b>  | <b>SDCNV7</b> | skip if zero                                                                                                                               |
|               | <b>MVI</b> | <b>C,'0'</b>  | otherwise replace the blank in C with an ASCII zero                                                                                        |
|               | <b>XAF</b> |               | and see if the sign has already been placed in                                                                                             |

|               |             |               |                           |
|---------------|-------------|---------------|---------------------------|
|               | <b>ORA</b>  | <b>A</b>      | the string                |
|               |             |               | if it has been zeroed the |
|               | <b>BZ</b>   | <b>SDCNV6</b> | sign is already in place  |
|               | <b>SAX</b>  | <b>-1</b>     | yes, it has               |
|               |             |               | no it hasn't. Place the   |
|               |             |               | sign in the word imme-    |
|               |             |               | diately preceding the     |
|               |             |               | address where the cur-    |
|               |             |               | rent digit will be placed |
|               | <b>ZAR</b>  |               | and clear the ASCII       |
|               |             |               | sign                      |
| <b>SDCNV6</b> | <b>XAF</b>  |               | restore the digit         |
| <b>SDCNV7</b> | <b>ORA</b>  | <b>C</b>      | OR with blank or          |
|               |             |               | ASCII zero                |
|               | <b>SAX</b>  | <b>0</b>      | place digit in string     |
|               | <b>INX</b>  | <b>X</b>      | increment for next digit  |
|               | <b>DBNZ</b> | <b>SDCNV3</b> | and get next digit        |
|               | <b>LAX</b>  | <b>-1</b>     | make sure we have at      |
|               |             |               | least one                 |
|               | <b>ORI</b>  | <b>:B0</b>    | significant numeric       |
|               |             |               | digit                     |
|               | <b>SAX</b>  | <b>-1</b>     | as before                 |
|               | <b>RET</b>  |               | finished, return to       |
|               |             |               | calling program           |

The table of powers of ten is exactly as it appears in example 13-7.

Conversion of binary fractions to decimal is much easier. The fraction is multiplied by ten, yielding an integer digit. This integer digit is the highest order decimal digit of the fraction. The integer digit is cleared and the remaining fraction again multiplied by ten, yielding another integer digit. This integer digit is the second decimal fraction digit. This process is carried on for as many places as required or until the fraction becomes zero. To see this clearly, consider a double precision number with the point between the halves of the number, i.e., with scale factor 8:

$$00000000.11000000_2 = .75_{10}$$

Treating this as a pure binary integer its value would be 192. Multiplying this number by ten gives:

$$00000111.10000000_2$$

The high order part of the number is a BCD 7, the high decimal digit. This integer is cleared to give:

$$00000000.10000000_2$$

and again multiplied by ten, to give:

$$00000101.00000000_2$$

The integer part of the product is a BCD 5, the second digit. The fraction has now become zero, so there is no more conversion to be done. The digits 7 and 5 are the decimal representation of the binary fraction .11000000.

A conversion of this type is shown in single precision in example 13-9.

### Example 13-9

A subroutine is required which will convert and print the eight bit binary fraction in A on entry. The conversion should end when there are no more significant digits to be generated, i.e., when the fraction becomes zero. Print a decimal point ahead of the decimal fraction.

| <i>label</i> | <i>inst.</i> | <i>operand</i> |                             |
|--------------|--------------|----------------|-----------------------------|
| SFC          | MOV          | L,A            | save original fraction in L |
|              | MVI          | A,'.'          | ASCII point to A            |
|              | CALL         | TTYOUT         | print the point             |
| SFC1         | MVI          | H,0            | clear upper product         |



|             |               |                        |
|-------------|---------------|------------------------|
| <b>DAD</b>  | <b>H</b>      | shift left once        |
| <b>MOV</b>  | <b>C,L</b>    | save shifted number    |
| <b>MOV</b>  | <b>B,H</b>    | ...                    |
| <b>DAD</b>  | <b>H</b>      | shift twice more       |
| <b>DAD</b>  | <b>H</b>      | ...                    |
| <b>DAD</b>  | <b>B</b>      | form final product, HL |
|             |               | times ten              |
| <b>MVI</b>  | <b>A,'0'</b>  | ASCII zero to A        |
| <b>ORA</b>  | <b>H</b>      | merge BCD integer with |
|             |               | ASCII digit frame      |
| <b>CALL</b> | <b>TTYOUT</b> | print the digit        |
| <b>MOV</b>  | <b>A,L</b>    | check remaining        |
|             |               | fraction for zero      |
| <b>ORA</b>  | <b>A</b>      | ...                    |
| <b>RZ</b>   |               | return if zero,        |
|             |               | conversion complete    |
| <b>BRA</b>  | <b>SFC1</b>   | otherwise convert      |
|             |               | another digit          |

Note that while eight bits is the equivalent of between two and three decimal digits precision, this does not guarantee that the generated decimal fraction will be this long. The binary fraction .00000001 will convert to its true decimal value, .00390625.

The process of fraction conversion in double precision is shown in example 13-10.

#### Example 13-10

A double precision 16 bit unsigned fraction is in HL. Convert the fraction to decimal, printing the successive digits. Presume the decimal point to have already been printed when the subroutine is entered.

| <i>label</i> | <i>inst.</i> | <i>operand</i> |                          |
|--------------|--------------|----------------|--------------------------|
| <b>DFC</b>   | <b>ZAR</b>   |                | clear A for triple shift |
|              | <b>DAD</b>   | <b>H</b>       | double shift fraction    |

|             |               |                         |
|-------------|---------------|-------------------------|
| <b>RAL</b>  |               | roll carry into A       |
| <b>MOV</b>  | <b>C,A</b>    | copy high order to C    |
| <b>MOV</b>  | <b>D,H</b>    | and lower orders to DE  |
| <b>MOV</b>  | <b>E,L</b>    | ...                     |
| <b>DAD</b>  | <b>H</b>      | shift left double again |
| <b>RAL</b>  |               | roll in bit shifted out |
| <b>DAD</b>  | <b>H</b>      | and again               |
| <b>RAL</b>  |               | ...                     |
| <b>DAD</b>  | <b>D</b>      | form low sum            |
| <b>ADC</b>  | <b>C</b>      | form high sum,          |
|             |               | propagate carry         |
| <b>ORI</b>  | <b>:B0</b>    | merge ASCII code bits   |
| <b>CALL</b> | <b>TTYOUT</b> | print the digit         |
| <b>MOV</b>  | <b>A,H</b>    | test fraction for zero  |
| <b>ORA</b>  | <b>L</b>      | zero yet?               |
| <b>RZ</b>   |               | yes, return to calling  |
|             |               | program                 |
| <b>BRA</b>  | <b>DFC</b>    | no, go generate another |
|             |               | digit                   |

Again, the reader is reminded that the equivalence of 16 bits to about 5 decimal digits in precision does *not* mean that this conversion will produce only 5 digits. The full decimal value of the binary fraction:

.11111111 11111111<sub>2</sub>

is:

.9999847412109375<sub>10</sub>

If this is likely to be bothersome set up a limiting digit counter whose expiration terminates the conversion whether or not the residual fraction has become zero. This could easily be done for the routine in example 13-10 by loading this count into B which is unused in the example and using **DBNZ** at the end of the loop to limit the number of digits generated.

Conversions between binary and octal are quite straightforward. Converting from binary to octal involves simply ORing :B0 onto each group of three bits. Conversion from octal to binary is done by checking that the ASCII digit is indeed legal octal, i.e., that it falls in the 0 - 7 range, then stripping the upper four bits from the ASCII code with an AND. The previous result is then shifted left three bits and the new digit ORed in.

Conversion between binary and hexadecimal is a little more difficult, because the character A does not immediately follow the character 9 in the ASCII sequence. This requires that the check for validity of an input hexadecimal character check against both the 0 - 9 and A - F ranges. If the character is in the A - F range it can be reduced to the proper binary sequence by subtracting seven, then removing the upper four bits with an AND.

Conversion from binary to hexadecimal requires that the 4 bit group to be converted to a hexadecimal character be tested to see if it is greater than 9. If so a 7 is added to it. When this is done adding :B0 to the result will give a correct hexadecimal result. The conversion from hexadecimal ASCII to binary is shown in the **FHCH** routine in the debugging monitor in appendix B. The opposite conversion is shown in the **CNV2HX** routine.

Another type of conversion, between decimal and floating point, will be taken up as part of the description of floating point in chapter 14.

## 14 / FLOATING POINT ARITHMETIC

“I was gratified to be able to answer promptly,  
and I did. I said I didn’t know.”

*Life on the Mississippi*  
Mark Twain

In situations in which scaling numbers is awkward or impossible, or in which the magnitudes of intermediate results cannot be known beforehand, resort must be taken to a system of arithmetic in which scale factors are kept automatically. This system is known as *floating point arithmetic*.

A floating point number resembles a number written in scientific notation in that it is made up of a magnitude that contains a point at some agreed upon place, and an exponent, i.e., a power of the number base, by which this magnitude is to be multiplied to yield the true value of the number. The value of:

$$2.99 \times 10^9$$

is thus:

$$2,990,000,000$$

The same value expressed as a decimal floating point number is

$$.299 \times 10^{10}$$

In a floating point number the magnitude is expressed as pure fraction. This number might be represented internally as:

| <i>sign</i> | <i>exponent</i> | <i>fraction</i> |
|-------------|-----------------|-----------------|
| +           | 10              | .2990           |
|             |                 | 235             |

if a four digit fraction were used. In a real decimal floating point system eight or more digits would be used, but for illustration we will keep it short. The sign appended to the floating point number above is the algebraic sign of the number, not the exponent. Some form of exponent sign is required, however, to represent numbers which have negative exponents, like .001.

This is accomplished by adding a constant, known as a *bias*, to the exponent. In decimal systems with a two digit exponent this bias is 50. An indicated exponent of 50 thus means a real exponent of zero. The number  $2.99 \times 10^{10}$  would be written:

|             |                 |                 |
|-------------|-----------------|-----------------|
| <i>sign</i> | <i>exponent</i> | <i>fraction</i> |
| +           | 60              | .2990           |

the real exponent being arrived at by subtracting the bias, in this case 50, from the given exponent of 60. The above number could be written with equal numerical validity:

|   |    |       |
|---|----|-------|
| + | 61 | .0299 |
|---|----|-------|

A number in this form is said to be *unnormalized* by virtue of the fact that its highest fraction digit is a zero. In general, floating point operations are performed only on *normalized* numbers, i.e., on numbers with a significant digit in the highest fraction position. An unnormalized number may be normalized by shifting the fraction left, decrementing the exponent once for each left shift necessary to bring a significant digit into the highest fraction position.

Binary floating point numbers are represented in an exactly analogous way. The usual practice is to allocate four or five eight bit words to a floating point number. The commonest practice is to use four and this form will be illustrated here. The four words are allocated as follows. One bit is used for the algebraic sign, eight bits for the exponent and the remaining 23 for the fraction. The range of the eight bit exponent is divided into two parts by using a bias of  $128_{10}$  ( $80_{16}$ ). The number one in this binary floating point form looks like this:

|             |                 |                          |
|-------------|-----------------|--------------------------|
| <i>sign</i> | <i>exponent</i> | <i>fraction</i>          |
| 0           | 10000001        | .10000000000000000000000 |



The meaning of the sign bit is the same as in normal fixed point, zero for plus and one for minus. The true value of the exponent is found by subtracting the bias of 10000000 from the given exponent of 10000001 to give 1. The fraction value is  $.1_2$  or  $.5_{10}$ . The value of the number is thus  $.5 \times 2^1 = 1$ . The number -5 is represented:

1      10000011      .101000000000000000000000

The real exponent is 3, and the value of the fraction is  $5/8$ . The number is therefore  $-5/8 \times 2^3$  or  $-5/8 \times 8 = -5$ . The number one fourth is represented:

0      01111111      .100000000000000000000000

The true exponent is  $01111111_2 - 10000000_2$  or -1.

The real value of the number is thus  $.5 \times 2^{-1}$  or .25.

Floating point numbers are added or subtracted by first equalizing their exponents, then adding the fractions. The exponents serve exactly the same function as the scale factors in fixed point arithmetic. They are equalized by shifting the fraction of the number with the smaller exponent right by a number of places equal to the difference of the exponents. The fractions are then added or subtracted. If the sum of like signed numbers carries out of the high position the entire fraction is shifted right to bring the carried out bit back in, the low bit being lost in the process, and the exponent incremented to compensate. Addition of numbers of unlike sign involves first equalizing the exponents as explained above, then subtracting the fraction of the number with the minus sign from the other. This is a subtraction of one unsigned magnitude from another. If the result is negative, as indicated by a borrow out of the high order, the fraction is two's complemented and the sign set to negative. Overflow can be ignored during this negation, since the high bit of the high order fraction is a magnitude bit, not a sign. The subtraction of the fractions opens the possibility that the difference may have leading nonsignificant zeros, creating an unnormalized number. This is corrected by shifting the fraction left until a one occupies the high bit, decrementing the result exponent by one for each shift.

Floating point arithmetic in microcomputers is usually done by interpretive subroutines, though some hardware for the purpose is becoming available. It is well beyond the scope of a book like this to develop a full floating point subroutine package. Some idea of how a simple case of floating addition works can be had by studying example 14-1.

#### Example 14-1

A pair of like signed floating point numbers is contained in the Z80's registers. The high two orders of the augend fraction are in HL, the low order in D and the exponent in E, in the biased form discussed above. The addend is contained, in identical format, in the alternate or prime register set. Find the floating point sum of the two numbers, leaving this sum in the registers in the same format.

| <i>label</i> | <i>inst.</i> | <i>operand</i> |                                                                                                                                                                                                                                                                                                                                                                                  |
|--------------|--------------|----------------|----------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------|
|              | <b>MOV</b>   | <b>A,E</b>     | get augend exponent                                                                                                                                                                                                                                                                                                                                                              |
|              | <b>EXX</b>   |                | swap to access addend<br>in prime registers                                                                                                                                                                                                                                                                                                                                      |
|              | <b>SUB</b>   | <b>E</b>       | form difference of<br>exponents. If this difference is zero the exponents were equal and no shift is necessary. If it is positive and nonzero, as indicated by the absence of carry, the number with the smaller exponent is already in the addressable register set. If it is negative, as indicated by carry the number with the smaller exponent is in the other register set |

|            |             |            |
|------------|-------------|------------|
|            | <b>BZ</b>   | <b>ADD</b> |
|            | <b>BNC</b>  | <b>SHF</b> |
|            | <b>EXX</b>  |            |
|            | <b>NEG</b>  |            |
| <b>SHF</b> | <b>LRH</b>  |            |
|            | <b>RLR</b>  |            |
|            | <b>RDR</b>  |            |
|            | <b>DCR</b>  | <b>A</b>   |
|            | <b>BNZ</b>  | <b>SHF</b> |
| <b>ADD</b> | <b>PUSH</b> | <b>H</b>   |
|            | <b>MVI</b>  | <b>E,0</b> |

no shift if exponents equal  
 shift this number  
 shift the other one  
 negate for positive shift count  
 logical right shift high order fraction. When control reaches this point the number with the smallest exponent is in the addressable register. It is this number we want to shift  
 rotate bit into L from H  
 rotate bit into D from L  
 decrement shift count and do it again if necessary. When this loop is finished the scale factors of the numbers have been equalized and the addition can be done  
 push high order fraction onto stack. This is so it can be retrieved after the registers have been swapped again  
 clear smaller exponent. The result exponent is the larger one attached to the other number. The smaller one is cleared so it will not interfere with the low

**PUSH D****EXX****POP B****XCHG****DAD B****XCHG****POP B****DAC B****BNC EOA****RHR****RLR****RDR**

order addition which is about to take place push low order onto stack  
 swap to address other number  
 pop low order addend into B  
 low order augend to HL form low order sum. Note that the result exponent, i.e., the larger of the two original exponents, is still in E and undisturbed because the smaller exponent was set to zero above low order sum and exponent to D and E, high order augend to HL  
 high order addend to B from stack  
 form high order sum plus carry from low order. If a carry results from this addition a sum greater than 24 bits resulted.  
 finished if no carry otherwise bring the carry back into the high order result  
 moving the entire fraction right  
 ...

|     |     |   |                          |
|-----|-----|---|--------------------------|
|     | INR | E | and increment the result |
|     |     |   | exponent                 |
| EOA | ... |   | program continues        |

Example 14-1 shows an extremely simple case of floating point addition. It is intended as an illustration only. It does not address some problems which can arise in floating addition, the most important of which is known as *floating overflow*. Floating overflow occurs when the capacity of the eight bit exponent is exceeded. There is nothing to be done about this except to go to a floating point form which has greater exponent capacity. In some systems, the floating point processor substitutes, without notice to the user, the largest possible floating point number. Beware of this practice. Find out how the system you are using treats this case before you get mysterious results.

The generation of a number which is too small to be represented is known as *floating underflow*. The usual practice in this case is to set the result to zero, again with no notice to the user. Again, find out what your particular system does in this case. Setting the result to zero is usually the best route to follow, but will produce a surprise if the result is to be used as a divisor later.

There are two principal limitations to floating point arithmetic, speed and precision. If the floating point is done in interpretive subroutines as illustrated here, a simple addition or subtraction will require dozens or even hundreds of computer instructions. There is no way to avoid this, but the process can be optimized by "tight" programming practices. A great deal of ingenuity has been expended on this subject by many people. One area which is subject to improvement is the shifting required to equalize the exponents. In example 14-1 this was done by a simple repetitive shift and count down. This works but is very slow. A better way would be to eliminate shifting in multiples of eight bits by moving entire words until the count had been reduced to less than eight. This process is shown in example 14-2.



## Example 14-2

A floating point number is in the registers in the same format as in example 14-1, with the shift count contained in A, as at location SHF of that example. Perform the shift in multiples of eight bits where possible, finishing the count by conventional shifting. The C register is available for use. Presume that the zero difference case has already been eliminated.

| <i>label</i> | <i>inst.</i> | <i>operand</i> |                                                                                                                    |
|--------------|--------------|----------------|--------------------------------------------------------------------------------------------------------------------|
| SHF          | MVI          | C,0            | clear C                                                                                                            |
| SHF2         | SUI          | 8              | reduce shift count by 8                                                                                            |
|              | JM           | SHF3           | skip out if less than zero                                                                                         |
|              | MOV          | D,L            | shift the entire fraction by 8 bits                                                                                |
|              | MOV          | L,H            | former high order now second order                                                                                 |
|              | MOV          | H,C            | and clear high order                                                                                               |
|              | BRA          | SHF2           | and try again                                                                                                      |
| SHF3         | ADI          | 8              | restore count. If the result here is a zero the shift count was an integral multiple of 8 and the shifting is done |
|              | BZ           | SHF5           | skip if zero                                                                                                       |
| SHF4         | LRH          |                | otherwise complete with conventional shift                                                                         |
|              | RLR          |                |                                                                                                                    |
|              | RDR          |                |                                                                                                                    |
|              | DCR          | A              |                                                                                                                    |
|              | BNZ          | SHF4           |                                                                                                                    |
| SHF5         | ...          |                | program continues                                                                                                  |

While some of the deficit in speed can be made up by careful programming, the loss of precision in floating point operations is another matter. The principal reason for the precision loss in addition and

subtraction is the shifting required to equalize the exponents. As the number with the smaller exponent is shifted right, the bits shifted out are simply lost, the result being that the floating point sum is somewhat smaller than the true sum. The amount of the loss varies with the length of the fraction and the difference between the exponents. If the exponent difference is greater than or equal to the number of fraction bits the effect is to zero the number being shifted. A floating add in these circumstances effectively becomes a rather elaborate do-nothing instruction. If possible, therefore, it will pay to add floating point numbers in such a way as to minimize this shifting, i.e., to form the sum in such a way that the exponent of the next number to be added is as nearly equal to the exponent of the sum already developed as possible. Effectively, this means adding the numbers from smallest to largest, rather than largest to smallest.

The effect of this precision loss is best seen in a decimal illustration. Let's go back for a moment to the decimal floating point system discussed at the beginning of this chapter, using a two digit exponent and a four digit fraction. Consider the following three numbers:

| <i>number</i> | <i>floating point</i> |
|---------------|-----------------------|
| 3.904         | 51 .3904              |
| 7.115         | 51 .7115              |
| 2.717         | 51 .2717              |

We will add these numbers in two different orders. First add the bottom two numbers:

$$\begin{array}{r}
 51 \quad .2717 \\
 + \quad 51 \quad .7115 \\
 \hline
 51 \quad .9832
 \end{array}$$

Since the exponents were the same, no pre-addition shift was necessary and the fractions could be added directly. Now add the top number to this sum:

$$\begin{array}{r}
 51 \quad .9832 \\
 + \quad 51 \quad .3904 \\
 \hline
 51 \quad 1.3736
 \end{array}$$

which is adjusted to normal floating point form by shifting the sum right to make it pure fraction and incrementing the exponent. This result is:

$$52 \quad .1373 \quad \text{or } 13.73$$

the final 6 being lost in the shift.

Now add the numbers in the other order, from top to bottom. Adding the top pair:

$$\begin{array}{r} 51 \quad .3904 \\ + \quad 51 \quad .7115 \\ \hline 51 \quad 1.1019 \end{array}$$

which is adjusted to:

$$52 \quad .1101$$

the 9 digit being lost in the shift process. Now we add the third number:

$$\begin{array}{r} 52 \quad .1101 \\ + \quad 51 \quad .2717 \\ \hline \end{array}$$

This addition cannot be done in this form because the exponents are not equal. They are equalized by shifting the fraction of the bottom number right one place and incrementing the exponent. The second number now looks like this:

$$52 \quad .0271$$

the 7 having been lost because of the shift, forming the sum:

$$\begin{array}{r} 52 \quad .1101 \\ + \quad 52 \quad .0271 \\ \hline 52 \quad .1372 \quad \text{or } 13.72 \end{array}$$

compared to 13.73 when the numbers were added in the other order.

Neither sum is correct, the true value being 13.736, but the first value is much closer. It is easy to see why this happened. In the first addition, the sum did not cause the exponent value to change, so that the third number could be added without a pre-addition shift. This caused an overflow which had to be corrected by shifting the result right but only the last digit of the final sum was lost in the shift. In the second case the first addition caused the exponent to change, losing the low digit in the compensating shift. Even more important, this made the exponents unequal when the third number was added, forcing a pre-addition shift. The low digit of the third number was thus lost before the addition had taken place, resulting in a smaller sum.

The same effect appears in binary floating point, of course. The effect is intrinsic to floating point methods, not any number system. This kind of precision loss can be minimized, but never eliminated. There are a couple of approaches which can be taken to cut the loss. The first is the use of a so-called "guard digit". Using the guard digit approach, the bits shifted out during the pre-addition shift are saved and shifted back in if the result requires normalization, i.e., if leading zeros were generated in the fraction by the operation. This helps a little, a very little.

Another approach to the problem is known as *noisy mode*. In the absence of a guard digit, bits shifted out during exponent equalization are simply lost. If post normalization is required, zero bits are shifted in. Using noisy mode ones can be shifted in instead. With zeros shifted in the sum will always be too small. With ones shifted in it will be too large. The problem can be run in both modes and an idea of the precision loss gained by comparing the results. This approach to the problem of floating point precision loss was the one taken on the IBM 7030 of the late 1950's, the legendary "STRETCH" machine. STRETCH took an even more realistic approach to floating point loss, the only really effective approach. It used a 64 bit word, allowing a fraction length which would satisfy the precision requirements of all but the pickiest theoreticians. Such machines are, unfortunately, long gone, and their loss is mourned.

Floating multiplication and division are, by contrast, simpler than addition and subtraction. The product is formed by multiplying the two fractions. The result exponent is formed by adding the exponents of multiplier and multiplicand and subtracting the bias. If the

multiplication of the fractions yielded a product whose highest bit was zero, e.g.  $.1000 \dots$  times  $.1000 \dots$ , the result fraction is shifted left and the exponent decremented to give a normalized result. Division is equally straightforward. The exponent of the quotient is the difference of the exponents of dividend and divisor. The dividend fraction is divided by the divisor fraction, to yield the result fraction. If the division is impossible because the dividend fraction is greater than or equal to the divisor fraction, the divisor fraction is shifted right one bit and the result exponent incremented to compensate. If the divisor has already been verified to be nonzero, this guarantees divisibility, since the dividend fraction will now have a zero in the high bit position while the normalized divisor fraction has a one in this position. The actual division of the fractions is done by the method described at the end of chapter 8.

The eight bit exponent used here allows numbers in the range  $10^{\pm 38}$  to be represented. Where greater range is required a variant floating point scheme is sometimes used which achieves greater range in the same overall space, but further sacrifices precision in doing so. This scheme involves the use of a *true* hexadecimal exponent. The exponent, while written as a binary number, is interpreted as a power of 16, not 2. The fraction is still binary. The implication of this structure is that a change of 1 in the exponent yields a change of a factor of 16 in the value of the number. What this means in terms of lost precision is not so obvious. A typical implementation of this floating point form uses one bit for the sign as usual, 7 bits for the exponent, and 24 for the fraction. The number one in this form looks like this, omitting the sign:

| <i>exponent</i> | <i>fraction</i>           |
|-----------------|---------------------------|
| 1000001         | .000100000000000000000000 |

The effective value of the exponent is one after removing the bias, meaning  $16^1$  not  $2^1$  as we have experienced before. The value of the necessary fraction thus becomes  $1/16$ , not  $1/2$ , i.e.,  $1 = 1/16 \times 16^1$ . This means that the number cannot be normalized as the leading zeros must be present. For the number one this makes no difference, but for a number involving a repeating binary fraction, like  $1.1_{10}$ , the precision loss due to the leading zero bits is serious. The overall effect is that this form of floating point gives only 21 guaranteed bits



of precision, even though the nominal fraction length is 24 bits.

Floating point arithmetic, then, is a trade of precision and speed for range. If the trade is necessary make it, but only if it is necessary. Floating point arithmetic is a useful, legitimate tool which has a place in the programmer's lexicon, but be careful with it and above all be aware of its limitations. No programmer who aspires to the effective use of floating point arithmetic should be without a copy of what has come to be the classic work in the area, *Rounding Errors in Algebraic Processes*, by J. H. Wilkinson (Prentice-Hall, 1963).

Conversion of floating point numbers between decimal and binary involves use of a method known as *radix deflation*. The general idea of the method is to eliminate the binary exponent in stages while building a decimal exponent. We will illustrate the case in which the true value of the exponent is positive and then generalize it. Take the floating point number:

|                 |                           |
|-----------------|---------------------------|
| <i>exponent</i> | <i>fraction</i>           |
| 10000111        | .111111100000000000000000 |

The true value of this exponent is 7, while the value of the fraction is  $127/128$ . The value of the number is therefore  $127/128 \times 2^7$  or  $127/128 \times 128 = 127$ . For clarity we will rewrite this in decimal and work through the example in decimal. The number is:

|                        |                   |                         |
|------------------------|-------------------|-------------------------|
| <i>binary exponent</i> | <i>fraction</i>   | <i>decimal exponent</i> |
|                        | $\frac{127}{128}$ |                         |
| 7                      |                   | 0                       |

We begin by subtracting 3 from the binary exponent. This has the effect of dividing the number by 8. In addition, we multiply the fraction by  $4/5$ . The total effect of these two steps is to divide the floating point number by ten. This entitles us to increment the decimal exponent by one. The number now looks like this:

|                        |                                      |                         |
|------------------------|--------------------------------------|-------------------------|
| <i>binary exponent</i> | <i>fraction</i>                      | <i>decimal exponent</i> |
|                        | $\frac{127}{128} \times \frac{4}{5}$ |                         |
| 4                      |                                      | 1                       |

since the binary exponent is still greater than 3 we repeat the process, subtracting 3 from the binary exponent, multiplying the fraction again by 4/5 and adding one to the decimal exponent. The result looks like this:

| <i>binary exponent</i> | <i>fraction</i>                                         | <i>decimal exponent</i> |
|------------------------|---------------------------------------------------------|-------------------------|
| 1                      | $\frac{127}{128} \times \frac{4}{5} \times \frac{4}{5}$ | 2                       |

Since we can no longer subtract 3 from the binary exponent without going negative, we stop here. The remaining 1 in the binary exponent can be disposed of by doubling the fraction. The result now looks like this:

| <i>binary exponent</i> | <i>fraction</i>                                                  | <i>decimal exponent</i> |
|------------------------|------------------------------------------------------------------|-------------------------|
| 0                      | $\frac{127}{128} \times \frac{4}{5} \times \frac{4}{5} \times 2$ | 2                       |

This last doubling of the binary fraction is done in the real conversion by a left shift. The binary exponent has now been gotten rid of. The value of the former fraction is:

$$\frac{127}{128} \times \frac{4}{5} \times \frac{4}{5} \times 2 = \frac{127}{100} = 1.27$$

This single integer digit and the fraction can be converted to decimal by the normal fixed point means. Attaching the generated decimal exponent we have:

$$1.27 \times 10^2$$

If the binary exponent is negative it is incremented by three and the fraction multiplied by 5/4 instead of 4/5. This effectively multiplies the floating point number by ten. The decimal exponent is then decremented by one. This process continues until the binary exponent is positive and less than 4. The residual binary exponent is then disposed of by a left shift of the fraction, producing a single integer digit and a binary fraction which are converted by the normal fixed

point means. The decimal exponent is then converted as a negative number and attached to the result.

Conversion from decimal to binary follows the same general plan. The decimal exponent is reduced (deflated) and a binary exponent built up.

There are some practical difficulties which arise with application of the radix deflation method. Perhaps the most significant is caused by the fact the  $4/5$  is not exactly representable as a binary fraction, i.e.:

$$\frac{4}{5} = .11001100110011001100 \dots_2$$

This introduces a loss of precision during the conversion, the amount of the loss depending on the magnitude of the number being converted, since for numbers with large positive binary exponents many iterations through the process, meaning many multiplications by  $4/5$ , will be necessary. There are a number of measures which will keep this error under control. First, the multiplication by  $4/5$  should be done in at least one word greater precision than that used for the binary fraction. This helps confine the error to the extra order which will not be converted. The error can be further reduced by reducing the number of multiplications. This is done by dividing by 100 or 1000 at a time rather than 10. To divide by 100 reduce the exponent by 6 instead of 3. This effectively divides by 64. The fraction is then multiplied by  $16/25$ , the square of  $4/5$  and the decimal exponent is incremented twice. The importance of this is that only one multiplication is done instead of two. Division by 1000 is done by reducing the binary exponent by 9, multiplying the fraction by  $64/125$ , the cube of  $4/5$ , and incrementing the decimal exponent by three. After every multiplication, the resulting fraction must be checked for normalization. If the high fraction bit has become a zero, the fraction should be shifted left and the binary exponent incremented. Keeping the fraction normalized in this way also helps to reduce the error.

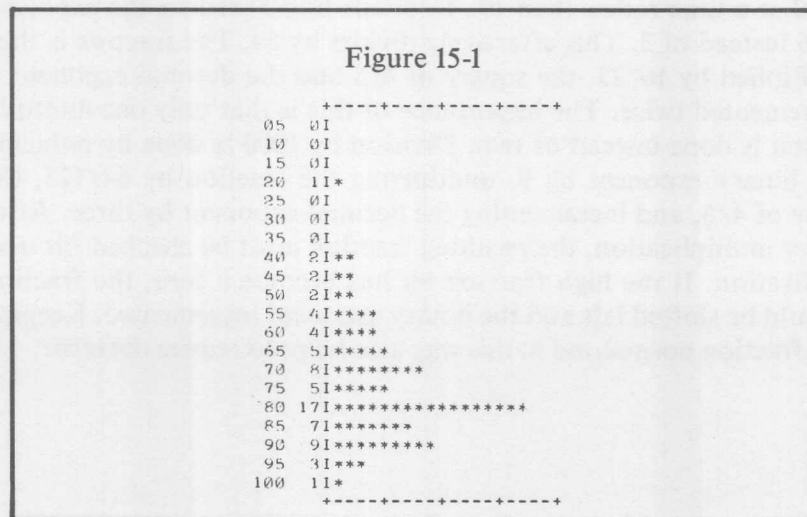
## 15 / GRAPHIC OUTPUT

“One picture is worth more  
than ten thousand words.”

Chinese Proverb

Graphic output of computed results allows a large amount of data to be displayed in an easily interpretable form, allowing trends and relations to be seen which might easily be obscured if the data were viewed in their raw numerical form. For this reason programmers, almost from the beginning of the craft, have sought to invent methods for displaying results in graphic form. It is the purpose of this chapter to show the elementary methods by which such data displays are constructed.

Perhaps the simplest type of graph is that shown in figure 15-1. This display shows a distribution of test scores, the length of each column indicating the number of scores which fell into the indicated interval. This type of display is known as a *histogram* or *frequency plot*.



The construction of such a display involves processing the raw scores into a table of frequencies. This table is simply an array in memory, the successive members of which contain the number of scores which fell into the interval indicated by the array member. In our example the scores are grouped into intervals of 5. The first frequency array member will thus contain the number of scores less than 5, the second the number 5 or greater but less than 10, etc. To build this table of frequencies the array of twenty memory words must first be cleared. The raw scores are then processed by dividing each by the interval size, in this case 5, yielding a quotient in the 0 - 19 range. The score 100 is forced into the 20th interval. This result is then used as a displacement to increment the appropriate member of the frequency table. The required address is computed by adding the displacement to the base address of the frequency table. The word at this computed address is then incremented and the entire process repeated for each score in the table of raw data.

The division by 5 can be done by a faster and more direct method than a full software divide using the expression:

$$N/5 = N/4 - N/16 + N/64 \dots$$

Since the denominators of these fractions are all powers of two the implied divisions can all be accomplished by right shifts. To preserve precision the above series is rewritten:

$$N/5 = \frac{N - N/4 + N/16}{4} \dots$$

the division by four being left to the last step.

The actual display is created by typing a constant character under control of a count, this count being the number of scores which fell into the interval represented by the line. Both the construction of the frequency table and the printing of the display are shown in example 15-1.

#### Example 15-1

A list of 70 test scores is in memory beginning at



**SCORES.** Create the program necessary to determine their frequency distribution and plot it as the histogram shown in figure 15-1. Space for the frequency array is created by:

```
label    inst.    operand
FARRAY RES    20
```

or

```
FARRAY BSS    20
```

The program to build the frequency table and plot the histogram is:

|             |             |                 |                                                                                                                       |
|-------------|-------------|-----------------|-----------------------------------------------------------------------------------------------------------------------|
| <b>FREQ</b> | <b>LXI</b>  | <b>H,FARRAY</b> | address of frequency array to H                                                                                       |
|             | <b>MOV</b>  | <b>D,H</b>      | copy to DE                                                                                                            |
|             | <b>MOV</b>  | <b>E,L</b>      | ...                                                                                                                   |
|             | <b>INX</b>  | <b>D</b>        | increment for move                                                                                                    |
|             | <b>LXI</b>  | <b>B,19</b>     | the number of necessary moves is one less than the actual space to be cleared                                         |
|             | <b>ZAR</b>  |                 |                                                                                                                       |
|             | <b>MOV</b>  | <b>M,A</b>      | clear the first array word                                                                                            |
|             | <b>MW+R</b> |                 | clear the array                                                                                                       |
|             | <b>LXI</b>  | <b>X,SCORES</b> | address of raw score array to X                                                                                       |
|             | <b>MVI</b>  | <b>B,70</b>     | number of raw scores to B                                                                                             |
|             | <b>LXI</b>  | <b>H,FARRAY</b> | address of array origin to HL. This is pushed onto the stack so it can be easily retrieved later without using an LXI |
|             | <b>PUSH</b> | <b>H</b>        |                                                                                                                       |

|     |      |      |                                                                                          |
|-----|------|------|------------------------------------------------------------------------------------------|
| FR1 | LAX  | 0    | pick up raw score                                                                        |
|     | INX  | X    | increment pointer                                                                        |
|     | MOV  | E,A  | now divide by 5, save original value                                                     |
|     | LRA  |      |                                                                                          |
|     | LRA  |      | N/4                                                                                      |
|     | MOV  | C,A  | save N/4                                                                                 |
|     | LRA  |      |                                                                                          |
|     | LRA  |      | N/16                                                                                     |
|     | ADD  | E    | add original value of N                                                                  |
|     | SUB  | C    | subtract N/4                                                                             |
|     | LRA  |      | now the final division by 4                                                              |
|     | LRA  |      | and we have N/5                                                                          |
|     | CPI  | 20   | was the original score 100                                                               |
|     | BLS  | FR2  | skip if not                                                                              |
| FR2 | MVI  | A,19 | otherwise force it into the last interval                                                |
|     | MOV  | E,A  | move computed displacement to E                                                          |
|     | MVI  | D,0  | clear upper half of displacement addend                                                  |
|     | POP  | H    | retrieve address of frequency table origin                                               |
|     | PUSH | H    | reset stack pointer for next fetch                                                       |
|     | DAD  | D    | form address of word in frequency table origin                                           |
|     | INR  | M    | and increment the memory word. Since there are only 70 total scores this cannot overflow |
|     | DBNZ | FR1  | and do it again                                                                          |
|     | POP  | H    | restore stack pointer                                                                    |
|     |      |      |                                                                                          |

|               |             |                 |                                                                                                                                                                                                                                           |
|---------------|-------------|-----------------|-------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------|
|               | <b>LXI</b>  | <b>H,BORDER</b> | print the dashed graph border                                                                                                                                                                                                             |
|               | <b>CALL</b> | <b>PRT</b>      |                                                                                                                                                                                                                                           |
|               | <b>LXI</b>  | <b>X,FARRAY</b> | address of array origin to X                                                                                                                                                                                                              |
|               | <b>MVI</b>  | <b>D,0</b>      | clear D for interval numbering                                                                                                                                                                                                            |
|               | <b>MVI</b>  | <b>B,20</b>     | line count to B                                                                                                                                                                                                                           |
| <b>PLOT1</b>  | <b>MVI</b>  | <b>A,:8D</b>    | return Teletype carriage                                                                                                                                                                                                                  |
|               | <b>CALL</b> | <b>CHOUT</b>    | ...                                                                                                                                                                                                                                       |
|               | <b>MVI</b>  | <b>A,:8A</b>    | ...                                                                                                                                                                                                                                       |
|               | <b>CALL</b> | <b>CHOUT</b>    | ...                                                                                                                                                                                                                                       |
|               | <b>MOV</b>  | <b>A,D</b>      | get interval                                                                                                                                                                                                                              |
|               | <b>ADI</b>  | <b>5</b>        | increment interval by 5                                                                                                                                                                                                                   |
|               | <b>DAA</b>  |                 | decimal adjust result                                                                                                                                                                                                                     |
|               | <b>MOV</b>  | <b>D,A</b>      | save for next pass                                                                                                                                                                                                                        |
|               | <b>MVI</b>  | <b>A,' '</b>    | blank to A for first character of line except if the <b>DAA</b> caused a carry. If this was the case, the result of the <b>DAA</b> had to be 100, i.e., 00 and a carry. The first character in this case must be a one instead of a blank |
|               | <b>BNC</b>  | <b>PLOT1A</b>   | print first character, whatever it is                                                                                                                                                                                                     |
|               | <b>MVI</b>  | <b>A,'1'</b>    |                                                                                                                                                                                                                                           |
| <b>PLOT1A</b> | <b>CALL</b> | <b>CHOUT</b>    | print first character, whatever it is                                                                                                                                                                                                     |
|               | <b>MOV</b>  | <b>A,D</b>      | get two interval digits                                                                                                                                                                                                                   |
|               | <b>RAR</b>  |                 | move high BCD digit to low four bits                                                                                                                                                                                                      |
|               | <b>RAR</b>  |                 |                                                                                                                                                                                                                                           |
|               | <b>RAR</b>  |                 |                                                                                                                                                                                                                                           |
|               | <b>RAR</b>  |                 |                                                                                                                                                                                                                                           |
|               | <b>ANI</b>  | <b>:F</b>       | clear upper four digits                                                                                                                                                                                                                   |
|               | <b>ORI</b>  | <b>'0'</b>      | form ASCII character                                                                                                                                                                                                                      |

|              |             |              |                                                                                                                                                                                                                                           |
|--------------|-------------|--------------|-------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------|
|              | <b>CALL</b> | <b>CHOUT</b> | print high ASCII<br>interval digit                                                                                                                                                                                                        |
|              | <b>MOV</b>  | <b>A,D</b>   | get interval digits again                                                                                                                                                                                                                 |
|              | <b>ANI</b>  | <b>:F</b>    | clear high BCD digit                                                                                                                                                                                                                      |
|              | <b>ORI</b>  | <b>'0'</b>   | form low ASCII digit                                                                                                                                                                                                                      |
|              | <b>CALL</b> | <b>CHOUT</b> | print it                                                                                                                                                                                                                                  |
|              | <b>MVI</b>  | <b>A,' '</b> | print blank between<br>interval and left border<br>of graph                                                                                                                                                                               |
|              | <b>CALL</b> | <b>CHOUT</b> | ...                                                                                                                                                                                                                                       |
|              | <b>LAX</b>  | <b>0</b>     | get count in this interval                                                                                                                                                                                                                |
|              | <b>MVI</b>  | <b>C,'0'</b> | digit frame to C                                                                                                                                                                                                                          |
| <b>PLOT2</b> | <b>SUI</b>  | <b>10</b>    | convert count                                                                                                                                                                                                                             |
|              | <b>BC</b>   | <b>PLOT3</b> | ...                                                                                                                                                                                                                                       |
|              | <b>INR</b>  | <b>C</b>     | ...                                                                                                                                                                                                                                       |
|              | <b>BRA</b>  | <b>PLOT2</b> | ...                                                                                                                                                                                                                                       |
| <b>PLOT3</b> | <b>ADI</b>  | <b>:BA</b>   | this add takes the place<br>of both the <b>ADI 10</b><br>which would have been<br>necessary to restore the<br>number to its condition<br>before the last subtraction<br>and the <b>ORI '0'</b><br>required to form the<br>ASCII character |
|              | <b>PUSH</b> | <b>PSW</b>   | save units digit                                                                                                                                                                                                                          |
|              | <b>MOV</b>  | <b>A,C</b>   | retrieve tens digit                                                                                                                                                                                                                       |
|              | <b>CPI</b>  | <b>'0'</b>   | leading digit zero?                                                                                                                                                                                                                       |
|              | <b>BNE</b>  | <b>PLOT4</b> | skip if not                                                                                                                                                                                                                               |
|              | <b>MVI</b>  | <b>A,' '</b> | substitute blank                                                                                                                                                                                                                          |
| <b>PLOT4</b> | <b>CALL</b> | <b>CHOUT</b> | print tens digit                                                                                                                                                                                                                          |
|              | <b>POP</b>  | <b>PSW</b>   | retrieve units digit from<br>stack                                                                                                                                                                                                        |
|              | <b>CALL</b> | <b>CHOUT</b> | print it                                                                                                                                                                                                                                  |
|              | <b>MVI</b>  | <b>A,'I'</b> | print border                                                                                                                                                                                                                              |
|              | <b>CALL</b> | <b>CHOUT</b> | character                                                                                                                                                                                                                                 |

|              |             |                 |                                                                                                                           |
|--------------|-------------|-----------------|---------------------------------------------------------------------------------------------------------------------------|
|              | <b>LAX</b>  | <b>0</b>        | get count in interval<br>again                                                                                            |
|              | <b>INX</b>  | <b>X</b>        | bump pointer up                                                                                                           |
|              | <b>ORA</b>  | <b>A</b>        | count zero?                                                                                                               |
|              | <b>BZ</b>   | <b>PLOT 6</b>   | anything to print? If so                                                                                                  |
|              | <b>MOV</b>  | <b>C,A</b>      | move count to C. The<br>count will be used to<br>control the length of the<br>string of characters<br>(asterisks) printed |
|              | <b>MVI</b>  | <b>A,'*</b>     | plotting character to A                                                                                                   |
| <b>PLOT5</b> | <b>CALL</b> | <b>CHOUT</b>    | print character                                                                                                           |
|              | <b>DCR</b>  | <b>C</b>        | decrement count                                                                                                           |
|              | <b>BNZ</b>  | <b>PLOT5</b>    | do it again if count not<br>zero                                                                                          |
| <b>PLOT6</b> | <b>DBNZ</b> | <b>PLOT1</b>    | decrement interval<br>count, do it again if not<br>zero                                                                   |
|              | <b>LXI</b>  | <b>H,BORDER</b> | print lower border of<br>graph                                                                                            |
|              | <b>CALL</b> | <b>PRT</b>      |                                                                                                                           |
|              | <b>...</b>  |                 | program continues                                                                                                         |

The ASCII string for the upper and lower borders of the graph is created by:

```
BORDER  ASC  '  +----+----+----+----+'
          DATA :8D
```

This last carriage return (:8D) is required as the terminator by the PRT routine, as previously discussed.



A more common plotting requirement involves plotting the value of one variable against another, commonly called an X-Y plot. This type of plot is usually done by construction of an image of the graph in memory. A space known as the *image region* is allotted in memory and set to ASCII blanks. The list of points to be plotted is then scanned and an address in the image region computed for each point, this address corresponding to the eventual position of that point on the plot. The blank at this location is then replaced by whatever character has been chosen to represent the plotted points.

The fact that the printing of the graph takes place from top to bottom makes the construction of the image a bit awkward, since the points with the highest Y value occupy the memory locations with the lowest addresses. The adjustment for this is fairly simple. The Y value of the point to be plotted is first checked for range, to see if it falls within the borders of the graph. If it is in range the maximum possible value for Y, 40 in the case of the example to follow, is subtracted from the Y value of the point to be plotted. This gives a zero or negative value which is then negated to yield a line number in the 0 through N-1 range, where N is the number of lines in the graph. This number is then multiplied by the width, in characters, of the graph. The X value of the point, after being checked for range, is then added to this number. This gives the address of the memory word which represents that point on the graph, relative to the beginning of the image region. The addition of the address of the first word of the image region to this relative address completes the necessary computation. The result is the address of the memory word into which the plotting character must be placed. This process is then repeated for each point to be plotted.

The printing of the finished graph is then a straightforward process, the result being shown in figure 15-2, which is a 40 by 40 first quadrant X-Y plot. The disproportion of the axes was necessary to keep equal resolution in both X and Y, since a Teletype prints 10 characters per inch horizontally but only 6 per inch vertically.

The upper border of the graph is first printed as in example 15-1. The contents of the image region memory locations are then printed in groups of 40 (in this case, since the plot is 40 wide) preceded and followed by whatever vertical border material is required. The lower border is then repeated to finish the graph. The details of this type of plot are shown in example 15-2.

## Example 15-2

Construct a program which will plot and type out a 40 by 40 X-Y plot as in figure 15-11. The X data are in an array which begins at location X, and the Y data are in an equal length array beginning at Y. The array length, N, has been defined elsewhere with an EQU.

| <i>label</i>  | <i>inst.</i> | <i>operand</i> |                                                                                                                                                                                                                                                                    |
|---------------|--------------|----------------|--------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------|
| <b>XYPLOT</b> | <b>LXI</b>   | <b>H,IMAGE</b> | address of image region                                                                                                                                                                                                                                            |
|               | <b>LXI</b>   | <b>X,X</b>     | address of X data to X register                                                                                                                                                                                                                                    |
|               | <b>LXI</b>   | <b>Y,Y</b>     | address of Y data to Y register. Note that the names of the X and Y data arrays do not conflict with the names of the registers. The assembler knows the difference. Memory locations may be given any legal name. There are no reserved symbols in this assembler |
|               | <b>MOV</b>   | <b>D,H</b>     | copy image address to DE                                                                                                                                                                                                                                           |
|               | <b>MOV</b>   | <b>E,L</b>     | ...                                                                                                                                                                                                                                                                |
|               | <b>INX</b>   | <b>D</b>       | increment "to" address for move                                                                                                                                                                                                                                    |
|               | <b>MVI</b>   | <b>A,' '</b>   | ASCII blank to A                                                                                                                                                                                                                                                   |
|               | <b>LXI</b>   | <b>B,1599</b>  | the reserved space for a 40 by 40 image is 1600 words. The number of moves will be one less than this                                                                                                                                                              |
|               | <b>MOV</b>   | <b>M,A</b>     | set first word of image to blank                                                                                                                                                                                                                                   |

|           |                |  |                                                                                                                                                      |
|-----------|----------------|--|------------------------------------------------------------------------------------------------------------------------------------------------------|
|           | <b>MW+R</b>    |  | set image region to<br>blanks                                                                                                                        |
|           | <b>MVI B,N</b> |  | count of points to be<br>plotted to B                                                                                                                |
|           | <b>MVI C,1</b> |  | the C register will be<br>used as a counter to<br>determine the interval<br>at which the "tick"<br>mark should be inserted<br>in the vertical border |
| <b>P1</b> | <b>LAX 0</b>   |  | get X coordinate of<br>point                                                                                                                         |
|           | <b>CPI 40</b>  |  | out of range?                                                                                                                                        |
|           | <b>BGE OUT</b> |  | yes if greater than 39                                                                                                                               |
|           | <b>LAY 0</b>   |  | get Y coordinate                                                                                                                                     |
|           | <b>CPI 40</b>  |  | test for range                                                                                                                                       |
|           | <b>BGE OUT</b> |  | ...                                                                                                                                                  |
|           | <b>SUI 39</b>  |  | if Y is in the 0 - 39 range                                                                                                                          |
|           | <b>NEG</b>     |  | subtract the vertical<br>bias, then negate                                                                                                           |
|           | <b>MOV L,A</b> |  | move to L and zero H in<br>preparation                                                                                                               |
|           | <b>MVI H,0</b> |  | for multiplication by<br>graph width                                                                                                                 |
|           | <b>DAD H</b>   |  | since $40 = 32 + 8$ we<br>can shift left                                                                                                             |
|           | <b>DAD H</b>   |  | 3 times to multiply by 8,                                                                                                                            |
|           | <b>DAD H</b>   |  | then twice more to<br>multiply by 32, and add<br>save in DE                                                                                          |
|           | <b>MOV D,H</b> |  | ...                                                                                                                                                  |
|           | <b>MOV E,L</b> |  | ...                                                                                                                                                  |
|           | <b>DAD H</b>   |  | ...                                                                                                                                                  |
|           | <b>DAD H</b>   |  | 40 times the "inverted"                                                                                                                              |
|           | <b>DAD D</b>   |  | Y coordinate in DE                                                                                                                                   |
|           | <b>LEX 0</b>   |  | X coordinate to E<br>register                                                                                                                        |

|            |             |                 |                                                                                                                                                                 |
|------------|-------------|-----------------|-----------------------------------------------------------------------------------------------------------------------------------------------------------------|
|            | <b>MVI</b>  | <b>D,0</b>      | clear D for double addition                                                                                                                                     |
|            | <b>DAD</b>  | <b>D</b>        | HL now contains 40 times the Y displacement plus the X coordinate. This is the address of the point to be plotted relative to the beginning of the image region |
|            | <b>LXI</b>  | <b>D,IMAGE</b>  | now add the address of the first word of the image region to form the memory address of the point to be plotted.                                                |
|            | <b>DAD</b>  | <b>D</b>        | plotting character to A place plotting character in memory word                                                                                                 |
|            | <b>MVI</b>  | <b>A,'*</b>     |                                                                                                                                                                 |
|            | <b>MOV</b>  | <b>M,A</b>      |                                                                                                                                                                 |
| <b>OUT</b> | <b>INX</b>  | <b>X</b>        | increment X pointer                                                                                                                                             |
|            | <b>INX</b>  | <b>Y</b>        | increment Y pointer                                                                                                                                             |
|            | <b>DBNZ</b> | <b>P1</b>       | and repeat until B is zero                                                                                                                                      |
|            | <b>LXI</b>  | <b>H,BORDER</b> | print top graph border                                                                                                                                          |
|            | <b>CALL</b> | <b>PRT</b>      | ...                                                                                                                                                             |
|            | <b>LXI</b>  | <b>H,IMAGE</b>  | address of completed image region to HL                                                                                                                         |
|            | <b>MVI</b>  | <b>D,40</b>     | line count to D                                                                                                                                                 |
| <b>P2</b>  | <b>MVI</b>  | <b>B,40</b>     | line width to B                                                                                                                                                 |
|            | <b>MVI</b>  | <b>A,:8D</b>    | return Teletype carriage                                                                                                                                        |
|            | <b>CALL</b> | <b>CHOUT</b>    | ...                                                                                                                                                             |
|            | <b>MVI</b>  | <b>A,:8A</b>    | ...                                                                                                                                                             |
|            | <b>CALL</b> | <b>CHOUT</b>    | ...                                                                                                                                                             |
|            | <b>MVI</b>  | <b>A,'I'</b>    | get border character                                                                                                                                            |
|            | <b>DCR</b>  | <b>C</b>        | time for "tick" mark?                                                                                                                                           |
|            | <b>BNZ</b>  | <b>P2A</b>      | not if C isn't zero yet                                                                                                                                         |
|            | <b>MVI</b>  | <b>C,5</b>      | yes, reload C with count                                                                                                                                        |

|            |             |                 |                         |
|------------|-------------|-----------------|-------------------------|
| <b>P2A</b> | <b>MVI</b>  | <b>A,'+'</b>    | and get "tick" mark     |
|            | <b>PUSH</b> | <b>PSW</b>      | save on stack for other |
|            |             |                 | end of line             |
| <b>P3</b>  | <b>CALL</b> | <b>CHOUT</b>    | print border character  |
|            | <b>MOV</b>  | <b>A,M</b>      | get image character     |
|            | <b>INX</b>  | <b>H</b>        | bump pointer            |
|            | <b>CALL</b> | <b>CHOUT</b>    | print image character   |
|            | <b>DBNZ</b> | <b>P3</b>       | end of line yet?        |
|            | <b>POP</b>  | <b>PSW</b>      | yes, retrieve border    |
|            |             |                 | character               |
|            | <b>CALL</b> | <b>CHOUT</b>    | and print it            |
|            | <b>DCR</b>  | <b>D</b>        | last line?              |
|            | <b>BNZ</b>  | <b>P2</b>       | not until D is zero     |
|            | <b>LXI</b>  | <b>H,BORDER</b> | print lower graph       |
|            |             |                 | border                  |
|            | <b>CALL</b> | <b>PRT</b>      | ...                     |

The top and bottom border lines of the graph are defined:

```

BORDER ASC    ' + ---- + ---- + ---- + ---- + '
ASC          ' ---- + ---- + ---- + ---- + '
DATA         :8D

```

and the image region:

```

IMAGE  RES    1600

```

As in example 15-1, the **PRT** and **CHOUT** routines are those previously illustrated.

Example 15-2 raises an interesting possibility. Since all of the plotted points of the graph except the border are either blank or asterisk, they could be represented by single one or zero bits. This would allow a single bit to represent a graph point rather than an entire word, resulting in a much smaller memory requirement for the internal graph image. The required memory image space for the graph in ex-



ample 15-2 would be 200 words instead of the 1600 used. This would require the manipulation of individual bits in memory, a task for which the Z80 is equipped with special instructions.

Any bit in an 8 bit register or a memory word may be individually tested, set to one, or cleared to zero by means of special Z80 bit manipulation instructions. The general mnemonic for testing bits in any register is **BTr**, in which the small **r** is the name of an 8 bit register. The bits are considered to be numbered starting with zero from the right, the leftmost bit being bit 7. The result of a **BTr** instruction is the setting of the Z flag in the F register to reflect the outcome of the test. The sequence:

```

BTC      3
BZ       ZILCH

```

will transfer control to location **ZILCH** if bit 3, i.e., the fourth bit from the right in register C is zero. Similarly, the sequence:

```

BTL      0
BNZ      ODD

```

will send control to location **ODD** if the lowest bit, bit zero, of the L register is a one, i.e., if the L register contains an odd number. Bits in a memory word pointed to by HL may be tested by a sequence like:

```

BTM      7
BNZ      MINUS

```

will test the high bit of the word whose address is contained in HL and send control to **MINUS** if the high bit, i.e., the sign bit, is a one.

A bit in a register or memory may be forced to one by the **BSr** instruction, the small **r** again being the name of an eight bit register or an **M**, the **M** meaning the memory word pointed to by HL. The instruction:

```

BSB      0

```

will force the low bit, bit 0, of the B register to one. This facility may be used to implement a true arithmetic left shift on the Z80. In this type of shift the seven magnitude bits are shifted left, introducing a zero in the rightmost bit, but leaving the sign unchanged. An arithmetic left shift is accomplished by:

|            |            |                               |
|------------|------------|-------------------------------|
| <b>ANI</b> | <b>:BF</b> | clear highest magnitude bit   |
| <b>LLA</b> |            | sign bit to carry, shift left |
| <b>BNC</b> | <b>S2</b>  | skip if it was positive       |
| <b>BSA</b> | <b>7</b>   | otherwise set sign negative   |
| ...        |            |                               |

Note that this procedure does not detect the loss of a significant bit through the shift. Note also that the SLA mnemonic used by Zilog for arithmetic left shift does not in fact perform an arithmetic shift but a logical shift. This is equivalent to the **LLA** instruction in our assembler. Bits in memory may be forced to ones by the **BSM** instruction. This forces the bit named in the operand of the **BSM** in the memory word pointed to by HL to be forced to one.

Bits in registers or memory may be cleared to zero by the **BCr** and **BCM** instructions, **r** again being the name of an eight bit register, and **M** meaning the location pointed to by HL.

The X and Y registers may also be used by the bit test, clear and set operations to point to their operands. The mnemonic changes in this case by appending the name of the pointer register. The instruction:

**BTMX 0,3**

means test bit 3 of the word pointed to by X, the displacement of the instruction being zero. The instruction:

**BSMY 3,5**

forces bit 5 of the word three beyond the one pointed to by Y to one, while

**BCMX -2,6**

clears the number 6 bit of the word two lower than that pointed to by X.

The principal weakness of the bit manipulation instructions is that, while the address of the operand word may be held in registers HL, X, or Y, the bit identification within the word is assembled as part of the instruction. Since placing points within an image in a plot routine would require that the bit address of the point be computed individually for each point, the instructions themselves would have to be modified. This makes the procedure very awkward and renders it useless for programs held in ROM. Perhaps in a future machine facility could be provided for holding the bit identification as well as the operand word address in a register. This would open up a large number of useful possibilities for the bit manipulation instructions. As these instructions stand, they are much less useful than they could be.

The final exercise in this chapter involves an extra twist on X-Y plotting, the addition of a true third dimension to a two dimensional graph. This is shown in figure 15-3. This plot is used by astronomers and is known as a Hertzsprung-Russell diagram, named for the astronomers who devised it. Its meaning is interesting enough to allow a short diversion into it. Those not interested in astronomy can skip over to the details of how the plot is actually done. The Hertzsprung-Russell diagram, or HR diagram for short, is a plot of the true brightnesses of stars against their colors. The vertical coordinate is absolute magnitude, the magnitude a star would display if it were at a standard distance of one parsec, about 3.26 light years. Magnitude is a logarithmic scale of brightness, a difference of 5 magnitudes meaning a difference of 100 in actual light energy received. The brighter a star is, the more negative its magnitude. Thus the star whose magnitude is -1 near the top of the graph is 100 times brighter than one whose magnitude is +4.

The horizontal coordinate is the spectral class of the star, a rough measure of its surface temperature. Stars at the left of the graph are blue or blue-white, the change in color progressing through yellow to red at the extreme right. The OBAFGKM classification across the top corresponds to a classification scheme in which

the temperatures of the stars may be roughly estimated from the appearance of their optical spectra.

The most interesting feature of the HR diagram is the broad diagonal stripe of stars from upper left to lower right. This group of stars is known as the *main sequence*, of which the sun is a member. Though it is not actually plotted here, the position of the sun in this scheme is at spectral class G2, just to the right of the center of the graph and absolute magnitude 4.8, about 2/3 of the way down from the top. The group of stars at the upper right, known as the giants, have evolved from stars which formerly occupied the upper end of the main sequence. The principal source of the vertical scatter in the diagram is the uncertainty in the distance measurements from which the absolute magnitudes are computed. Readers interested in further details of the HR diagram can find them in any modern text on astronomy or astrophysics.

The plotting of such a diagram involves several extra twists over a simple X-Y plot. The first is that some indication must be given of how many stars fell within a single diagram cell. This involves changing the plotting character to reflect this number. Second, the horizontal coordinate is alphabetic, and not an orderly alphabetic scheme at that. The alphabetic horizontal coordinate must be transformed into a numerical one to allow points to be plotted horizontally. Finally, the vertical coordinate is inverted, increasing downward rather than upward as with most plots. The plotting and printing of the Hertzsprung-Russell diagram in figure 15-3 is shown in example 15-3.

#### Example 15-3

Create the program which will construct the image of the Hertzsprung-Russell diagram in figure 15-3 and print it. The data for the diagram are in ASCII, five characters per star, in a single data array beginning at **SDATA**. The format of the five word entries in **SDATA** is:

1st 2 characters: spectral class in the form F2 or B9

3rd character: the algebraic sign of the absolute magnitude, - for negative and blank for positive.

4th and 5th characters: the numerical magnitude itself, with the decimal point omitted. Only the integer and first fraction digit are given.

In this format the data for the star at the upper left corner of the diagram are:

**ASC      'B9-10'**

while the data for the star at the lowest part of the diagram are:

**ASC      'K5 68'**

meaning spectral class B9, absolute magnitude -1.0 and spectral class K5 absolute magnitude +6.8. The program for the generation and printing of the diagram is:

| <i>label</i>  | <i>inst.</i> | <i>operand</i>   |                                                                                                                                     |
|---------------|--------------|------------------|-------------------------------------------------------------------------------------------------------------------------------------|
| <b>HRDGRM</b> | <b>LXI</b>   | <b>H,IMAGE</b>   | address of image region to HL                                                                                                       |
|               | <b>LXI</b>   | <b>D,IMAGE+1</b> | address of second word to DE                                                                                                        |
|               | <b>ZAR</b>   |                  | clear A                                                                                                                             |
|               | <b>MOV</b>   | <b>M,A</b>       | set 1st image word to zero                                                                                                          |
|               | <b>LXI</b>   | <b>B,3263</b>    | the image region is 51 lines by 64 characters or 3264 words. The number of moves required to zero the region is one less than this. |
|               | <b>MW+R</b>  |                  | clear the entire image to zero                                                                                                      |



|    |     |         |                                                                                                                       |
|----|-----|---------|-----------------------------------------------------------------------------------------------------------------------|
| H1 | LXI | X,SDATA | address of data array to X                                                                                            |
|    | LXI | B,N     | number of stars to B                                                                                                  |
|    | LAX | 0       | get alphabetic spectral class                                                                                         |
|    | MVI | D,0     | clear D to transform to number                                                                                        |
|    | CPI | 'O'     | was it spectral class O?<br>This is an "oh" not a zero                                                                |
|    | BEQ | H3      | skip if so                                                                                                            |
|    | MVI | D,10    | increment numerical class                                                                                             |
|    | CPI | 'B'     | spectral class B?                                                                                                     |
|    | BEQ | H3      | skip if so                                                                                                            |
|    | MVI | D,20    | increment numerical class                                                                                             |
|    | CPI | 'A'     | spectral class A?                                                                                                     |
|    | BEQ | H3      | ...                                                                                                                   |
|    | MVI | D,30    | ...                                                                                                                   |
|    | CPI | 'F'     | spectral class F                                                                                                      |
|    | BEQ | H3      | ...                                                                                                                   |
|    | MVI | D,40    | ...                                                                                                                   |
|    | CPI | 'G'     | spectral class G?                                                                                                     |
|    | BEQ | H3      | ...                                                                                                                   |
|    | MVI | D,50    | ...                                                                                                                   |
|    | CPI | 'K'     | spectral class K?                                                                                                     |
|    | BEQ | H3      | ...                                                                                                                   |
|    | MVI | D,60    | ...                                                                                                                   |
|    | CPI | 'M'     | spectral class M?                                                                                                     |
|    | BEQ | H3      | if control gets past here<br>the spectral class was<br>not one of the defined<br>ones and the star will be<br>ignored |
| H2 | JMP | NXT     | skip this star                                                                                                        |

|    |     |     |                                                                                                        |
|----|-----|-----|--------------------------------------------------------------------------------------------------------|
| H3 | LAX | 1   | pick up numerical subclass                                                                             |
|    | CPI | '0' | check numeric validity                                                                                 |
|    | BLS | H2  | skip if not numeric                                                                                    |
|    | CPI | :BA | greater than 9?                                                                                        |
|    | BGE | H2  | skip if so                                                                                             |
|    | ANI | :F  | valid, clear upper four bits                                                                           |
|    | ADD | D   | add transformed spectral class                                                                         |
|    | SUI | 6   | this 6 is subtracted because the classes O0 through O5 are not plotted. A now contains a number 0 - 63 |
|    | MOV | E,A | save in E                                                                                              |
|    | LAX | 2   | pick up sign of absolute magnitude                                                                     |
|    | MVI | D,1 | set flag in D                                                                                          |
|    | CPI | '-' | is absolute magnitude negative?                                                                        |
|    | BNE | H4  | skip if not                                                                                            |
|    | DCR | D   | otherwise set D to zero                                                                                |
| H4 | LAX | 3   | pick up integer absolute magnitude digit                                                               |
|    | CPI | '0' | check for numeric validity                                                                             |
|    | BLS | H2  | skip star if not numeric                                                                               |
|    | CPI | :BA | ...                                                                                                    |
|    | BGE | H2  | ...                                                                                                    |
|    | ANI | :F  | clear upper four bits                                                                                  |
|    | LLA |     | multiply it by ten                                                                                     |
|    | MOV | H,A | ...                                                                                                    |
|    | LLA |     | ...                                                                                                    |
|    | LLA |     | ...                                                                                                    |
|    | ADD | H   | ...                                                                                                    |

|           |            |            |                             |
|-----------|------------|------------|-----------------------------|
|           | <b>MOV</b> | <b>H,A</b> | save digit times ten in H   |
|           | <b>LAX</b> | <b>4</b>   | pick up absolute            |
|           |            |            | magnitude fraction          |
|           |            |            | digit                       |
|           | <b>CPI</b> | <b>'0'</b> | check numeric validity      |
|           | <b>BLS</b> | <b>H2</b>  | ...                         |
|           | <b>CPI</b> | <b>:BA</b> | ...                         |
|           | <b>BGE</b> | <b>H2</b>  | ...                         |
|           | <b>ANI</b> | <b>:F</b>  | clear upper four bits       |
|           | <b>ADD</b> | <b>H</b>   | add integer digit times     |
|           |            |            | ten                         |
|           | <b>DCR</b> | <b>D</b>   | test for negative           |
|           | <b>BZ</b>  | <b>H5</b>  | ...                         |
|           | <b>NEG</b> |            | negate binary               |
|           |            |            | magnitude                   |
| <b>H5</b> | <b>ADI</b> | <b>20</b>  | bias positive               |
|           | <b>JM</b>  | <b>NXT</b> | skip star if still negative |
|           | <b>ARA</b> |            | divide by two. This is      |
|           |            |            | done because the graph      |
|           |            |            | represents magnitude to     |
|           |            |            | 1/5 rather than 1/10 of     |
|           |            |            | a magnitude                 |
|           | <b>CPI</b> | <b>51</b>  | fall off bottom             |
|           |            |            | of graph?                   |
|           | <b>BGE</b> | <b>H2</b>  | skip if so                  |
|           | <b>MOV</b> | <b>L,A</b> | transfer to L               |
|           | <b>MVI</b> | <b>H,0</b> | clear upper register for    |
|           |            |            | multiply                    |
|           | <b>MOV</b> | <b>D,H</b> | clear upper half of DE.     |
|           |            |            | The horizontal graph        |
|           |            |            | displacement is still in E  |
|           | <b>DAD</b> | <b>H</b>   | multiply line number by     |
|           |            |            | 64                          |
|           | <b>DAD</b> | <b>H</b>   | ...                         |
|           | <b>DAD</b> | <b>H</b>   | ...                         |
|           | <b>DAD</b> | <b>H</b>   | ...                         |
|           | <b>DAD</b> | <b>H</b>   | ...                         |

|            |             |                 |                                                                                                                                                                  |
|------------|-------------|-----------------|------------------------------------------------------------------------------------------------------------------------------------------------------------------|
|            | <b>DAD</b>  | <b>H</b>        | ...                                                                                                                                                              |
|            | <b>DAD</b>  | <b>D</b>        | add horizontal displacement. At this point HL contains the address of the word to be incremented, relative to the first word of the image region                 |
|            | <b>LXI</b>  | <b>D,IMAGE</b>  | get address of image origin                                                                                                                                      |
|            | <b>DAD</b>  | <b>D</b>        | address of word to be incremented in HL                                                                                                                          |
|            | <b>INR</b>  | <b>M</b>        | increment the memory word. If this word was already FF <sub>16</sub> the result will be zero. This must be checked for and the cell set back to FF <sub>16</sub> |
|            | <b>BNZ</b>  | <b>NXT</b>      | skip if no zero                                                                                                                                                  |
|            | <b>DCR</b>  | <b>M</b>        | set count back                                                                                                                                                   |
| <b>NXT</b> | <b>LXI</b>  | <b>D,5</b>      | data array increment to DE                                                                                                                                       |
|            | <b>X+DE</b> |                 | add increment to data pointer. This is 5 because the data for each star requires 5 memory words                                                                  |
|            | <b>DCX</b>  | <b>B</b>        | decrement star count                                                                                                                                             |
|            | <b>MOV</b>  | <b>A,B</b>      | check for zero                                                                                                                                                   |
|            | <b>ORA</b>  | <b>C</b>        | ...                                                                                                                                                              |
|            | <b>JNZ</b>  | <b>H1</b>       | do it again if necessary                                                                                                                                         |
|            | <b>LXI</b>  | <b>H,STLINE</b> | print top spectral class line                                                                                                                                    |
|            | <b>CALL</b> | <b>PRT</b>      | ...                                                                                                                                                              |
|            | <b>LXI</b>  | <b>H,NLINE</b>  | print numerical subclass line                                                                                                                                    |
|            | <b>CALL</b> | <b>PRT</b>      |                                                                                                                                                                  |

|    |      |         |                                                                                                                          |
|----|------|---------|--------------------------------------------------------------------------------------------------------------------------|
| H6 | LXI  | X,IMAGE | address of image region to X                                                                                             |
|    | MVI  | D,1     | "tick" mark trigger to D                                                                                                 |
|    | MVI  | E,-2    | vertical magnitude marker to E                                                                                           |
|    | MVI  | C,51    | line count to C                                                                                                          |
|    | MVI  | A,:8D   | return Teletype carriage                                                                                                 |
|    | CALL | CHOUT   | ...                                                                                                                      |
|    | MVI  | A,:8A   | ...                                                                                                                      |
|    | CALL | CHOUT   | ...                                                                                                                      |
|    | MVI  | B,'I'   | border character to B                                                                                                    |
|    | DCR  | D       | time for "tick" mark?                                                                                                    |
|    | BNZ  | H10     | not if D didn't become zero                                                                                              |
|    | MVI  | D,5     | yes, reload D with count                                                                                                 |
|    | MVI  | B,'+'   | "tick" mark for border character                                                                                         |
|    | MOV  | A,E     | get magnitude                                                                                                            |
| H7 | INR  | E       | update magnitude                                                                                                         |
|    | ORA  | A       | set flags                                                                                                                |
|    | BNZ  | H7      | skip if not zero                                                                                                         |
|    | MVI  | H,' '   | no sign if zero                                                                                                          |
|    | BRA  | H9      | ...                                                                                                                      |
| H8 | JP   | H8      | skip if positive and nonzero                                                                                             |
|    | MVI  | H,'-'   | magnitude was negative                                                                                                   |
|    | NEG  |         | force positive                                                                                                           |
|    | BRA  | H9      | and skip                                                                                                                 |
|    | MVI  | H,'+'   | set sign positive                                                                                                        |
| H9 | ORI  | '0'     | merge ASCII zero. This can be done because the numerical magnitude never exceeds 8 due to the constraints of the diagram |



|      |      |       |                            |
|------|------|-------|----------------------------|
| H10  | MOV  | L,A   | save magnitude digit       |
|      | MOV  | A,H   | get ASCII sign/blank       |
|      | CALL | CHOUT | print it                   |
|      | MOV  | A,L   | get magnitude digit        |
|      | CALL | CHOUT | print it                   |
| H11  | BRA  | H11   | and skip                   |
|      | MVI  | A,' ' | no magnitude or "tick"     |
|      | CALL | CHOUT | mark                       |
| H12  | CALL | CHOUT | print blanks at left       |
|      | MVI  | A,' ' | of graph                   |
|      | CALL | CHOUT | ...                        |
|      | MOV  | A,B   | enter here from printing   |
|      | CALL | CHOUT | magnitude above            |
| H12A | PUSH | PSW   | print one blank            |
|      | MVI  | B,64  | get border character       |
|      | LAX  | 0     | print it                   |
|      | MVI  | H,' ' | save border character      |
|      | CPI  | 0     | for right end              |
| H12A | BEQ  | H14   | line length to B           |
|      | CPI  | 1     | get image word             |
|      | BNE  | H12A  | blank in case of zero      |
|      | MVI  | H,'*' | test for zero              |
|      | ZAR  |       | skip if zero               |
| H12A | BRA  | H14   | if only one star fell into |
|      | CPI  | 10    | this cell the plotting     |
|      | BGE  | H13   | character is an asterisk   |
|      | MVI  | H,'0' | skip if not one            |
|      |      |       | set asterisk as plotting   |
| H12A |      |       | character                  |
|      |      |       | and clear A for OR         |
|      |      |       | ...                        |
|      |      |       | if count was 10 or more    |
|      |      |       | use "#" as plotting        |
| H12A |      |       | character                  |
|      |      |       | ...                        |
|      |      |       | count between 2 and 9,     |
|      |      |       | use ASCII digit            |
|      |      |       |                            |

|            |             |                 |                                 |
|------------|-------------|-----------------|---------------------------------|
|            | <b>BRA</b>  | <b>H14</b>      | ...                             |
| <b>H13</b> | <b>ZAR</b>  |                 | clear A for OR                  |
|            | <b>MVI</b>  | <b>H,'#'</b>    | set plotting character to #     |
| <b>H14</b> | <b>ORA</b>  | <b>H</b>        | construct plotting character    |
|            | <b>CALL</b> | <b>CHOUT</b>    | print it                        |
|            | <b>INX</b>  | <b>X</b>        | increment pointer               |
|            | <b>DBNZ</b> | <b>H12</b>      | count down and do it again      |
|            | <b>POP</b>  | <b>PSW</b>      | retrieve border character       |
|            | <b>CALL</b> | <b>CHOUT</b>    | print it again                  |
|            | <b>DCR</b>  | <b>C</b>        | last line                       |
|            | <b>BNZ</b>  | <b>H6</b>       | not if still nonzero            |
|            | <b>LXI</b>  | <b>H,NLINE</b>  | print numerical subclass line   |
|            | <b>CALL</b> | <b>PRT</b>      | ...                             |
|            | <b>LXI</b>  | <b>H,STLINE</b> | print spectral class label line |
|            | <b>CALL</b> | <b>PRT</b>      | ...                             |

The spectral class labeling and numerical subclass lines are defined by:

|               |             |                                              |  |
|---------------|-------------|----------------------------------------------|--|
| <b>STLINE</b> | <b>RES</b>  | <b>4,' '</b>                                 |  |
|               | <b>ASC</b>  | <b>'0 B A F G'</b>                           |  |
|               | <b>ASC</b>  | <b>' K M'</b>                                |  |
|               | <b>DATA</b> | <b>:8D</b>                                   |  |
| <b>NLINE</b>  | <b>RES</b>  | <b>4,' '</b>                                 |  |
|               | <b>ASC</b>  | <b>'67890123456789012345678901234567890'</b> |  |
|               | <b>ASC</b>  | <b>'12345678901234567890123456789'</b>       |  |
|               | <b>DATA</b> | <b>:8D</b>                                   |  |

and the image region itself is defined:

|              |            |             |
|--------------|------------|-------------|
| <b>IMAGE</b> | <b>RES</b> | <b>3264</b> |
|--------------|------------|-------------|

There are, of course, almost as many possible forms of graphic output as there are types of data to be plotted. All plots will, however, employ the basic strategy of computing the address of a memory word to be set or incremented to form the final image. If the material in this chapter has been thoroughly understood, the reader should have no trouble constructing plot programs to serve any special purpose.

Figure 15-2

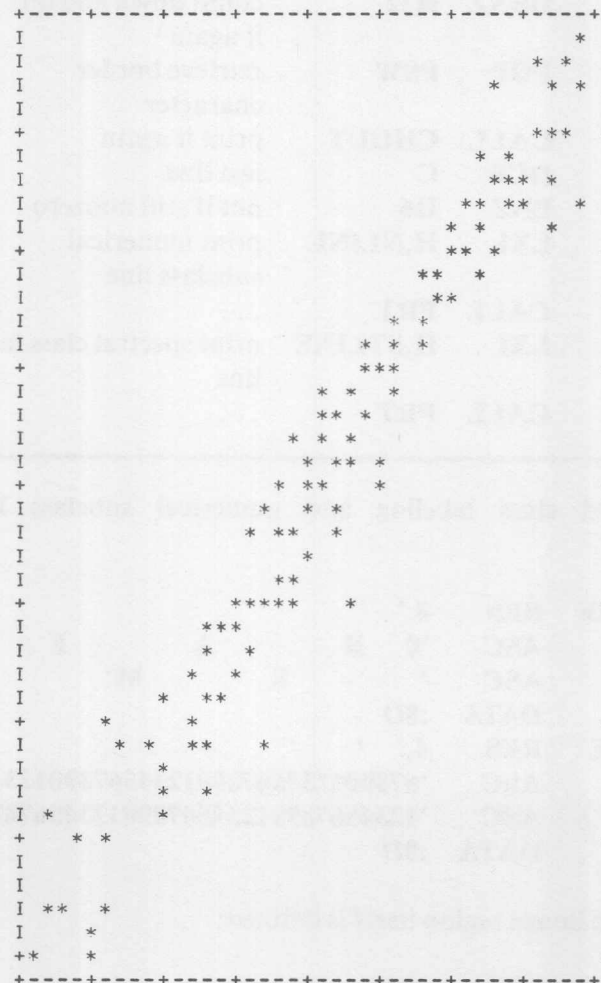
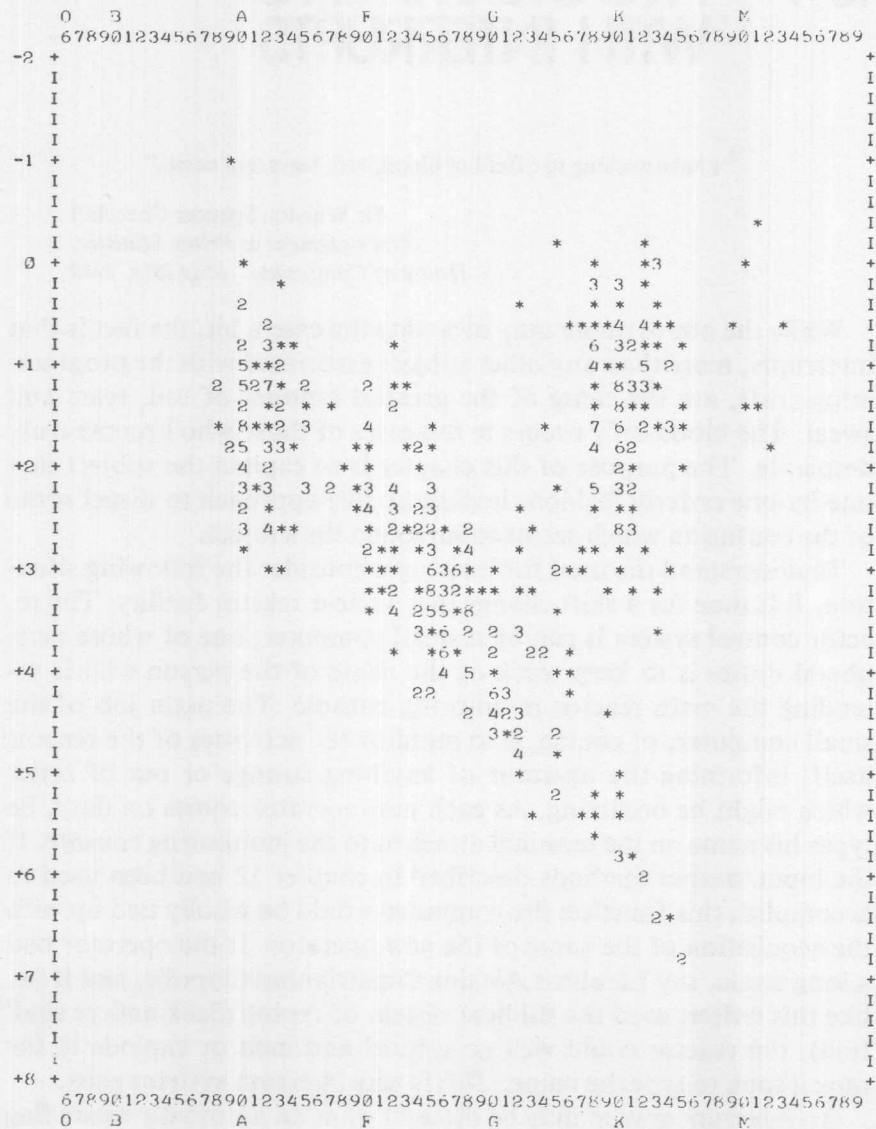


Figure 15-3



## 16 / PROGRAMMING WITH INTERRUPTS

“I have nothing to offer but blood, toil, tears and sweat.”

Sir Winston Spencer Churchill  
*First statement as Prime Minister,  
House of Commons, May 13th, 1940*

While the above quote may overstate the case a bit, the fact is that interrupts, more than any other subject associated with the programming craft, are the cause of the greatest amount of toil, tears and sweat. The blood only occurs in the cases of those who become truly desperate. The purpose of this chapter is to explain the subject in a one-by-one orderly fashion, hoping by this approach to dispel some of the confusion which seems to surround the subject.

To understand the need for interrupts consider the following situation. It is time for a shift change in a nuclear reactor facility. The reactor control system is run by a small computer, one of whose peripheral duties is to keep track of the name of the person who is attending the main reactor monitoring console. The main job of the small computer, of course, is to monitor the activities of the reactor itself, informing the operator of anything strange or out of order which might be occurring. As each new operator comes on duty, he types his name on the terminal attached to the monitoring console. If the input/output methods described in chapter 12 had been used to accomplish this function the computer would be wholly tied up with the acquisition of the name of the new operator. If the operator had a long name, say Lambros Alexios Papatriantaphilopolis, and if he, like this writer, used the Biblical system of typing (Seek and ye shall find), the reactor could well go critical and melt or explode in the time it took to type the name. *This* is why interrupt systems exist.

An interrupt system may be thought of as an automatic ready flag monitor which would allow the computer in the example above to devote its entire time to reactor monitoring, taking only the few



microseconds necessary to actually transfer data when a key was struck, then going back to the monitoring task. The business of waiting in a loop which reads the device status word, testing a bit and repeating the process until the bit changes state would be entirely eliminated. An interrupt system therefore allows the computer to perform tasks on demand, taking only as much time as the actual data transmission requires without idling while waiting for ready flags.

If there is a single concept which is most important to the understanding of interrupt service, it is this: *the interrupt service must in no way disrupt the task which is interrupted*. In the hardware this means first that no interrupt is acknowledged while an instruction is in the process of being executed. Interrupt acknowledgement can occur only *between* instructions, not during an instruction. It is fairly easy to see what would happen if this were not so. Suppose the computer were in a commercial environment, say typing paychecks, and were in the process of forming the final sum when an interrupt occurred. The interrupt service routine would destroy the partial result, leaving garbage in the registers when the interrupted task was resumed. We will return to this business of disruption of the interrupted task later, since there are other ways in which it can occur.

We will first detail here what happens during an interrupt, leaving the exact mechanics of how it occurs until a little later. When an interrupt occurs the following sequence of events transpires:

- 1) The instruction in progress is finished. Always.
- 2) Based on system wiring, a number is formed which is the memory address of the next instruction to be fetched.
- 3) The program counter is pushed onto the stack. Recall now the earlier and heavy emphasis which was laid on the fact that the program counter always points to the instruction following the one being executed. The address pushed onto the stack is therefore that of the instruction after the one in progress when the interrupt occurred.
- 4) The newly computed memory address, again based on system wiring for the particular device involved, is loaded into the program counter, effectively transferring control to this address.

- 5) Further interrupts are suppressed until again allowed by the interrupt service program.
- 6) When the interrupt service program has completed execution, its last act before returning control to the interrupted program is to again allow interrupts.
- 7) The top item on the stack, which will be the address of the instruction following the one after which the interrupt occurred (if there are no stack programming errors), is popped into the program counter, sending control back to the instruction following the interrupt point.

The above description is to be considered rough and preliminary, given in this form to furnish a jumping off point for the detailed discussion to follow. In particular, there are some simplifications made above which will be rectified further in this chapter.

We turn now to the specific conditions under which an interrupt can occur. As a base for discussion we will presume that the interrupting device is one of the terminals discussed in chapter 12. There are three general conditions which must be met before an interrupt can occur. These conditions are:

- 1) The interrupt system must be *enabled*. The term is specific to the interrupt system as a whole and has nothing to do with any specific peripheral device. It describes a CPU condition rather than a condition in a peripheral device or interface. The interrupt system is enabled by means of the **EI** (enable interrupts) instruction. If the CPU interrupt system has not been enabled it is said to be *disabled*. This term again applies to the CPU and has nothing to do with any peripheral device or interface. Once enabled the interrupt system may be disabled by means of the **DI** (disable interrupts) instruction. If this system is disabled no interrupt can occur under any circumstance except the so-called nonmaskable interrupt which will be discussed later.
- 2) The device interface must be *armed*. Arming is a condition of the device interface and not of the CPU. It consists of connecting the device ready flag to the interrupt line in such a way that the interrupt line is driven true when data are ready to be transmitted. If

the device interface is not armed, it is said to be *disarmed*. If the device interface is disarmed no interrupt from that device can occur, though interrupts may be generated by other devices whose interfaces are armed if the CPU interrupt system is enabled. The arming of the device interface is specific to that device. No general procedure for device arming and disarming can be given.

- 3) The ready flag in the device interface must enter the true state, i.e., indicate that the device is ready for data transmission or whatever other function is appropriate to the device.

If any of the above conditions is not met the interrupt does not occur.

The Z80 provides an unusually varied set of interrupt response possibilities, which we will take up one at a time. If no special action has been taken by the program or if an **IM0** instruction has been executed, the Z80 CPU functions in a mode known as interrupt mode 0. In this mode, the generation of an interrupt causes the CPU to request a response from the interrupting device interface. This response takes the form of an instruction, supplied by the device interface, and executed by the CPU. This instruction is *not* fetched from computer memory, but is generated by the peripheral device interface and supplied to the CPU. The instruction is normally one of eight special instructions reserved for this purpose, known as *restarts*. A restart instruction is a type of call, in that it forces the program counter to be pushed onto the stack. Unlike a call, however, a restart does not fetch its effective address from the memory locations following the instruction. The effective address of a restart is contained within the one word instruction itself. Bits 3, 4 and 5 of the restart instruction form a value which, when multiplied by eight, gives the address of the next instruction to be fetched after the program counter has been pushed onto the stack. This allows up to eight different sources of interrupt to generate distinct interrupt addresses, which can save considerable time in interrupt response.

The second Z80 interrupt response mode, called mode 1, entered by executing an **IM1** instruction forces all interrupts to location :38. This is a simple option if only one or a few devices are capable of interrupting and there is no particular time bind. If multiple sources of interrupt are used in mode 1, the interrupt service routine must test

each of the devices capable of interrupting until it finds the one which caused the interrupt. This is done just as in programmed input/output, by reading the status word of each device and testing the appropriate bit for the ready to transmit data condition. This process is known as *polling* and is one of the more common methods of interrupt service. Because each device must be individually tested for the ready to transmit state this method is slow, but offers the interesting possibility of establishing a software priority among the interrupting devices, by the sequence in which they are tested.

The final interrupt service mode of the Z80, mode 2, is entered by executing the **IM2** instruction. In interrupt mode 2 a special register known as the *Interrupt Vector Register* (IVR) comes into play. The interrupt vector register is loaded from the Z80 A register by the **TIA** (transfer to IVR from A) instruction. The contents of the IVR can be loaded into A by the **TAI** instruction. When an interrupt occurs in this mode, the CPU requests a response from the interrupting device interface. This response, an eight bit number the low bit of which *must* be zero, is used as the low eight bits of a memory address, the high eight bits being the contents of the IVR. The address thus generated and the one following it are then accessed to form yet a second address. This second address is the location of the interrupt service routine. This scheme allows the interrupt service locations to be anywhere in memory, in contrast to the eight lower memory locations of mode 0 and the single fixed location of mode 1.

In summary, mode 0 interrupt service forms an interrupt service address which is a multiple of 8, beginning at memory location 0, the highest address being :38 ( $56_{10}$  or  $7 \times 8$ ). This allows only eight words between the interrupt service addresses. Since a meaningful service routine can almost never be squeezed into such a small space the instruction at the interrupt service location will usually be a **JMP** to some higher location at which the service routine can be accommodated. In mode 1 all interrupts are forced to a common location, :38, and the service routine at that location has the responsibility of figuring out which device caused the interrupt. In mode 2, an address is computed using the contents of the IVR as the high eight bits and the device response as the low eight bits. The low bit of the device response *must* be zero. This address is used as a pointer to another address in memory. This second address is a pointer to the actual interrupt service routine. In all modes the eventual result of the interrupt

is to push the program counter onto the stack and begin execution at the computed interrupt service address. Simultaneously, the CPU interrupt system is disabled, so that no further interrupt can occur, except the nonmaskable interrupt, until an **EI** instruction has been executed.

For the vast majority of application cases, interrupt mode 0 will suffice. This mode mimics the action of the 8080 family machines and must be used if compatibility with the 8080 is to be preserved. As a matter of terminology, 'an interrupt system capable of generating distinct interrupt addresses for each interrupting device, as in modes 0 and 2 on the Z80, is known as a *vectored interrupt system*.

Now we must come to the specific details of interrupt service, and the obligations of the interrupt service routine. Just as the CPU hardware must acknowledge interrupts in such a way as not to disrupt the activity of the interrupted program, the interrupt service routine must do nothing to disrupt the interrupted program. Since the interrupt service routine uses the same registers as the program which is interrupted, it is the obligation of the interrupt service routine to save and restore all registers and flags which it uses. Further it must remove or neutralize the source of the interrupt so that when the interrupt system is reenabled another interrupt from the same source does not occur.

The great variety of possible interrupting devices makes specific discussion of arming sequences and interrupt source removal impossible. To include specific examples from the writer's system would only add confusion. A general approach will therefore be used to the examples which follow. Each of these has run on the writer's system, with the enabling and interrupt removal sequences appropriate to that system. The examples will be shown with the generally applicable service code intact, but with portions specific to this hardware configuration only indicated.

We will consider first the simplest possible kind of interrupt situation, that in which no data are to be transmitted. This type of interrupt occurs when the system is being used to count pulses from some source. The source may be external, as in counting cars entering an expressway ramp or shoppers breaking a photoelectric beam at a supermarket entrance, or it may be internal, as in counting crystal or line frequency generated pulses for the purpose of keeping time. No data are transmitted in these cases. The data are the interrupts them-



selves, the interrupt denoting the occurrence of some event. In this type of interrupt, the clearing of the flag which caused the interrupt is usually automatic, taking place as part of the interrupt acknowledge sequence.

A device which generates a train of pulses, evenly spaced in time, is known as a *real time clock*. It is in no sense a clock, of course, since it does not keep time at all, but we will respect the common nomenclature here. If time is to be kept by such a device, the keeping of it is done in software by the interrupt service routine. A real time clock service routine is shown in example 16-1. This example introduces new instructions. These instructions, their functions and mnemonics are:

**DCMX    d**

decrement the memory location pointed to by X plus the displacement d. The flags are set to reflect the result. The corresponding instruction for Y is **DCMY    d**

**SIX        d,n**

stores the eight bit immediate operand n in the memory location pointed to by X plus the displacement d. The corresponding instruction for Y is **SIY    d,n**. The immediate operand n may be of any legal 8 bit data type, for example, **SIX    1,'A'** stores the ASCII code for the letter A into the memory location one beyond that pointed to by X.

The use of these instructions to keep a time of day clock under interrupt control is shown in example 16-1.

## Example 16-1

An external source is generating an interrupting pulse at 120 Hz, twice the line frequency. The Z80 is operating in interrupt mode zero. The interrupting source will respond to the CPU acknowledgement of the interrupt with a restart instruction which causes control to be forced to memory location 8 (Restart 1). Presume that the device interface has already been armed and that locations 8,9 and A have been loaded with:

**JMP TICTOC**

Use the pulses to keep a software time of day clock. The clock cells are defined by:

| <i>label</i>  | <i>inst.</i> | <i>operand</i> |                                   |
|---------------|--------------|----------------|-----------------------------------|
| <b>PULSE</b>  | <b>DATA</b>  | <b>120</b>     | number of interrupt pulses/second |
| <b>SECOND</b> | <b>DATA</b>  | <b>60</b>      | seconds counter                   |
| <b>MINUTE</b> | <b>DATA</b>  | <b>60</b>      | minute counter                    |
| <b>HOURL</b>  | <b>DATA</b>  | <b>24</b>      | hour counter                      |

and the clock interrupt service routine is:

|               |             |                |                                                                                                                                                                                               |
|---------------|-------------|----------------|-----------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------|
| <b>TICTOC</b> | <b>PUSH</b> | <b>PSW</b>     | save A and the flags.<br>These are the same flags and accumulator as are used by the program which was interrupted and must be saved and restored to avoid disrupting the interrupted program |
|               | <b>PUSH</b> | <b>X</b>       | likewise for X                                                                                                                                                                                |
|               | <b>LXI</b>  | <b>X,PULSE</b> | address of pulse count to X                                                                                                                                                                   |

|               |                   |                                    |
|---------------|-------------------|------------------------------------|
|               | <b>DCMX 0</b>     | decrement pulse count              |
|               | <b>BNZ RETURN</b> | return if pulse count not yet zero |
|               | <b>SIX 0,120</b>  | reload pulse count                 |
|               | <b>DCMX 1</b>     | decrement seconds count            |
|               | <b>BNZ RETURN</b> | return if not yet zero             |
|               | <b>SIX 1,60</b>   | reload seconds counter             |
|               | <b>DCMX 2</b>     | decrement minute counter           |
|               | <b>BNZ RETURN</b> | return if not yet zero             |
|               | <b>SIX 2,60</b>   | reload minutes counter             |
|               | <b>DCMX 3</b>     | decrement hour counter             |
|               | <b>BNZ RETURN</b> | return if hours not yet zero       |
|               | <b>SIX 3,24</b>   | reload hours counter               |
| <b>RETURN</b> | <b>POP X</b>      | restore X register                 |
|               | <b>POP PSW</b>    | restore flags                      |
|               | <b>EI</b>         | REENABLE THE INTERRUPT SYSTEM!!!!  |
|               | <b>RET</b>        |                                    |

The service routine shown presumes that the flag which caused the clock interrupt was reset automatically. If this is not true in your particular configuration, the instruction necessary to clear this flag *must be included before the EI instruction is reached*. Failure to attend to this will lead to an immediate and total disaster. When interrupts are enabled by the **EI** the same interrupt will be waiting again. The interrupt will occur again and the clock will again be incremented. The computer is thus forced into a kind of interrupt loop from which it cannot escape.

The software clock kept by the above routine can be accessed for use by any other resident program by simply copying the contents of **SECOND**, **MINUTE** and **HOUR**, subtracting  $60_{10}$  from **SECOND** and **MINUTE** and  $24_{10}$  from **HOUR**. The resulting binary time inter-

vals can then be negated and converted to decimal for printing or other display. Note, they can be *converted*, not adjusted. Do not attempt to use **DAA**. It won't work. When copying the clock contents, the using program should disable interrupts for as long as required to make the copy. This is to prevent the clock changing during the copying process. If the clock reads 23 hours, 59 minutes, 59 seconds with a count of -1 in **PULSE** when a clock interrupt occurred the converted time would read 0 hours 59 minutes 59 seconds, nearly an hour wrong.

Before leaving example 16-1, a note must be made about the **EI** instruction. Suppose that the interrupt service routine in example 16-1 were counting pulses which arrived at irregular intervals, rather than 120 cycle pulse specified in the problem. The possibility now exists that a second pulse can arrive before the interrupt service routine has finished dealing with the first. If the enablement of interrupts occurred exactly when the **EI** instruction was executed the second interrupt would occur immediately after the **EI**, before the **RET**, i.e., the address of the "interrupted program" would be within the interrupt service routine itself. This address would be pushed onto the stack and the interrupt service routine again executed. If no more pulses arrived the **RET** would pop the return address from the stack into the PC. This return address is none other than the address of the **RET** itself. The **RET** would again be executed and control would be returned to the program which was originally interrupted. This is fine and the return is orderly. Picture however, what would happen if any significant number of interrupting pulses arrived at this higher frequency. Return address after return address would be pushed onto the stack before the computer had had the opportunity of disposing of the earlier ones. This could cause a very mysterious stack overflow. If the stack space were located between program segments the segment immediately below the stack space would suffer destruction. If the stack were in lower memory so that the overflow crossed address zero, the popping of return addresses would bring the contents of upper memory into the PC, if such memory exists, effectively transferring control to some unpredictable location. If address :FFFF did not exist in the machine the effect would be to pop the contents of a nonexistent memory location into PC. This would usually be all ones, and control would depart for that never-never land of nonexistent memory.

There is a simple solution to this problem. When an **EI** is executed, the enablement of the interrupt system is delayed until after the instruction which follows the **EI** has been completed. This instruction would normally be the **RET**. This procedure insures that control at least reaches the program which was originally interrupted before another interrupt can occur.

We turn now to another kind of interrupt situation, that in which data are to be transmitted. The interrupt in this case is simply a signal that the device is ready to transmit data. The effect of arming the device interface for a data transmitting device is to connect the ready flag, the same one we tested by reading the status word and shifting into carry in programmed input/output, to the interrupt line. This allows a transition of the ready flag from the not ready to the ready state to cause an interrupt, i.e., it completes the third of the three conditions for interrupt discussed before. Unlike the real time clock interrupt in example 16-1, however, the interrupt is not removed automatically. Since it was the ready flag which caused the interrupt, this flag must somehow be cleared, i.e., put into the not ready state. This removes the source of interrupt. This is done by the transmission of data. In the case of input, the **IN** or other input instruction clears the flag to the not ready state. For output, the transmission of data from the CPU to the peripheral interface by an **OUT** or other output instruction clears the ready flag. The input case is the simpler and will be considered first.

For terminal devices in general the input and output functions are considered separately, capable of being separately armed and disarmed. In many ways a terminal can be considered as two distinct devices. Given that the arming sequence for the device has already been executed, and the interrupt system enabled by an **EI**, the striking of a key will cause an interrupt. The data must then be read from the device to clear the ready flag. The basic sequence required for service of such an interrupt is shown in example 16-2.

#### Example 16-2

A terminal device connected to the Z80 has been wired so as to interrupt to location 24<sub>10</sub> when a key is struck. This location and the two following it have been loaded with:



**JMP   INSVCE**

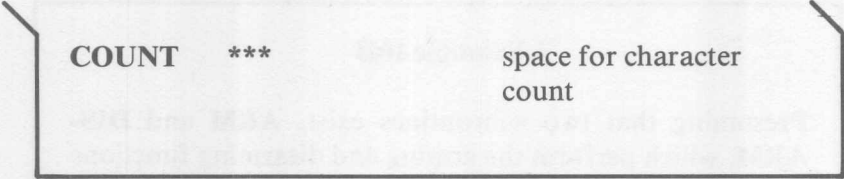
Given that the interface has already been armed, write a program which will read four characters under interrupt control, storing them in successive locations beginning at **NAME** and set the flag at **DONE** to a one when the fourth character has been read. The operating program has already executed the following subroutine:

|              |             |                                                               |
|--------------|-------------|---------------------------------------------------------------|
| <b>START</b> | <b>ZAR</b>  |                                                               |
|              | <b>STA</b>  | <b>DONE</b> clear <b>DONE</b> flag                            |
|              | <b>LXI</b>  | <b>H,NAME</b> address of beginning of <b>NAME</b> block to HL |
|              | <b>SHLD</b> | <b>PTR</b> save in pointer                                    |
|              | <b>MVI</b>  | <b>A,4</b> generate character count                           |
|              | <b>STA</b>  | <b>COUNT</b> ...                                              |
|              | <b>RET</b>  |                                                               |

The interrupt service routine is:

|               |                   |                                                                                                                                                                                                                                                                                      |
|---------------|-------------------|--------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------|
| <b>INSVCE</b> | <b>PUSH PSW</b>   | when the interrupt occurs control is transferred first to location 24 <sub>10</sub> , then here by the <b>JMP</b> at that address. By the time the <b>JMP</b> is executed the interrupt system has been automatically disabled, just as if a <b>DI</b> instruction had been executed |
|               | <b>PUSH H</b>     | save AF and HL                                                                                                                                                                                                                                                                       |
|               | <b>IN    DATA</b> | input from the terminal data address. This performs two functions. It brings the data into the                                                                                                                                                                                       |

|               |             |               |                                                                                                                                                                            |
|---------------|-------------|---------------|----------------------------------------------------------------------------------------------------------------------------------------------------------------------------|
|               | <b>ORI</b>  | <b>:80</b>    | A register <i>and</i> clears the ready flag in the terminal interface. It would be safe at this point to enable the interrupt system again, but we won't do this just yet. |
|               | <b>LHLD</b> | <b>PTR</b>    | force high bit on get pointer which was initialized by the <b>START</b> routine above                                                                                      |
|               | <b>MOV</b>  | <b>M,A</b>    | store the character just read                                                                                                                                              |
|               | <b>INX</b>  | <b>H</b>      | increment the pointer                                                                                                                                                      |
|               | <b>SHLD</b> | <b>PTR</b>    | replace pointer                                                                                                                                                            |
|               | <b>LDA</b>  | <b>COUNT</b>  | get character count                                                                                                                                                        |
|               | <b>DCR</b>  | <b>A</b>      | decrement it                                                                                                                                                               |
|               | <b>STA</b>  | <b>COUNT</b>  | replace count. Note that <b>STA</b> does not change the flags.                                                                                                             |
|               | <b>BNZ</b>  | <b>INSVC2</b> | skip if count not zero                                                                                                                                                     |
|               | <b>MVI</b>  | <b>A,1</b>    | time to set <b>DONE</b> flag                                                                                                                                               |
|               |             |               | if count zero                                                                                                                                                              |
|               | <b>STA</b>  | <b>DONE</b>   | set flag                                                                                                                                                                   |
| <b>INSVC2</b> | <b>POP</b>  | <b>H</b>      | restore HL                                                                                                                                                                 |
|               | <b>POP</b>  | <b>PSW</b>    | restore AF                                                                                                                                                                 |
|               | <b>EI</b>   |               | REENABLE THE INTERRUPT SYSTEM!!!!                                                                                                                                          |
|               | <b>RET</b>  |               | return to interrupted program                                                                                                                                              |
| <b>NAME</b>   | <b>RES</b>  | <b>4</b>      | block for storage of 4 characters                                                                                                                                          |
| <b>DONE</b>   | <b>***</b>  |               | <b>DONE</b> flag                                                                                                                                                           |
| <b>PTR</b>    | <b>DBL</b>  | <b>0</b>      | space for temporary pointer                                                                                                                                                |



COUNT \*\*\*                      space for character  
count

Note that the interrupt service examples have saved the registers and status of the interrupted program on the stack. It has been suggested in some literature that the alternate registers be reserved for the use of interrupt service programs, saving the time required for the **PUSHes** and **POP**s. This will work, but is counterproductive since it deprives the operating program of the use of these registers, effectively reducing the Z80 most of the way back to the 8080. In addition, it will not work for more than a single level of interrupts, precluding the use of a priority system.

The reader may have noticed by now that rather heavy emphasis is being laid on the reenabling of the interrupt system at the end of the interrupt service routine. This is because experience has shown that the failure to do so is one of the commonest of interrupt programming errors. Just why such a fuss should be made over interrupt programming errors in particular will become clear in a little bit.

The case of output interrupts is a little subtler. Recalling the definition of the ready condition, the state in which the device interface ready flag is true, if the output interrupt is armed, there will always be an interrupt pending when the device is idle. The only time no interrupt is pending is when the interface is busy processing a character. This requires a rather different approach to output interrupts. In general, the output interrupt must be kept disarmed when the interrupt system is enabled. It is armed only when data are to be transmitted and disarmed when the interrupt from the last transmitted word is received. This last interrupt *must* be dealt with and cannot be ignored as is suggested in some popular programming literature. To ignore this last interrupt in an output chain is to create the danger that the operating program will attempt to start another output chain before the last character of the previous chain has been completed, causing both to be garbled. An interrupt driven terminal output routine is shown in example 16-3.

## Example 16-3

Presuming that two subroutines exist, **ARM** and **DISARM**, which perform the arming and disarming functions for the terminal interface, create a string output routine which performs the same function as the **PRT** routine of example 12-4, but under interrupt control. The routine is to be used by the operating program by loading the address of the first character of the string into HL and **CALLing PRT**. The string comes to an end when a carriage return (:8D) is encountered as for the earlier **PRT** routine. Presume that the output interrupt is disarmed by a previous execution of **DISARM** when **PRT** is entered.

| <i>label</i> | <i>inst.</i> | <i>operand</i> |                                                                                                                                                                                                                                                                                                              |
|--------------|--------------|----------------|--------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------|
| <b>PRT</b>   | <b>SHLD</b>  | <b>APTR</b>    | enter here on call from operating program. Save pointer to first message character                                                                                                                                                                                                                           |
|              | <b>DI</b>    |                | disable the interrupt system                                                                                                                                                                                                                                                                                 |
|              | <b>CALL</b>  | <b>ARM</b>     | arm the output interrupt. With the interrupt system disabled nothing can happen now. Since the terminal output machinery is not occupied, however, the ready flag is in the true state, i.e., the terminal is ready to accept data and an interrupt will occur as soon as the system is enabled by <b>EI</b> |
|              | <b>EI</b>    |                | enable the system                                                                                                                                                                                                                                                                                            |
|              | <b>RET</b>   |                | and return. Since the actual enablement is de-                                                                                                                                                                                                                                                               |

layed until after the instruction following the **EI** is finished, control will return to the point after the **CALL** to **PRT** in the operating program. Since all three conditions for interrupt are fulfilled, interrupts enabled, device armed, and ready flag true an interrupt will occur as soon as the **RET** has been completed

The interrupt service routine is:

|               |                  |                         |
|---------------|------------------|-------------------------|
| <b>PRTSVC</b> | <b>PUSH H</b>    | save HL                 |
|               | <b>PUSH PSW</b>  | save AF                 |
|               | <b>LHLD APTR</b> | get character pointer   |
|               | <b>MOV A,M</b>   | get character in string |
|               | <b>INX H</b>     | and increment pointer   |
|               | <b>CPI :8D</b>   | was it carriage return? |
|               | <b>BEQ EPSVC</b> | skip if so              |
|               | <b>OUT DATA</b>  | otherwise send the      |
|               |                  | character to the termi- |
|               |                  | nal data port           |
|               | <b>SHLD APTR</b> | save pointer for next   |
|               |                  | pass                    |
| <b>RET</b>    | <b>POP PSW</b>   | restore the registers   |
|               |                  | used                    |
|               | <b>POP H</b>     | ...                     |
|               | <b>EI</b>        | REENABLE THE            |
|               |                  | INTERRUPT               |
|               |                  | SYSTEM!!!               |
|               | <b>RET</b>       |                         |



**EPSVC      CALL   DISARM**

at this point the last character has been processed and the output interrupt must be disarmed to prevent further meaningless interrupts caused by the now constant true state of the ready flag  
branch to common return  
space for address pointer

**BRA    RET**

**APTR      DBL    0**

Example 16-3 shows a procedure that will work if conditions guarantee that no request for output will ever occur while an output string is in progress. Consider what would happen if the operating program called **PRT** while terminal output was in progress. The current value of **APTR** would simply be destroyed by the new one. The remainder of the message in progress would be chopped off and the new message started immediately.

Clearly, this problem requires some solution. The solution is to create some kind of "lining" up queueing mechanism by which requests for output can be stacked up and serviced in an orderly way. In this case the interrupt service routine would be required to examine the queue to see if any output requests were pending. If anything were waiting to be processed, the new string address would be pulled from the queue and the first character sent out, instead of disarming the output interrupt as was done in example 16-3. The problem becomes a bit more complicated at this point. The general solution to it is not within the scope of the an elementary book like this.

The overall task of keeping an orderly flow of work going through a system which is responding to random external world demand is far from simple. Programs which operate in this mode are said to be *real time programs*. For systems of any significant complexity the problem is usually solved by the creation of a program which man-

ages the activities of application subprograms. A program capable of such management is known as a *real time executive* program. The structure of such a program will be the subject of a future volume in this series.

There remains to be discussed a special interrupt, known as the *nonmaskable interrupt*. This interrupt is not under control of **EI** and **DI** but is always enabled and has ultimate priority over any other CPU activity.

The use of this interrupt is usually reserved for some situation of more or less ultimate importance, a condition which cannot be ignored or deferred until other activities have been completed. In this category might fall such events as the onset of a power failure, the beginning of the battle of Armageddon or the start of the Monday night baseball game. The occurrence of the nonmaskable interrupt is deferred only until the instruction in progress has been completed. If another interrupt, one under **EI/DI** control is waiting, the nonmaskable interrupt is serviced first. Nothing has priority over it. The interrupt service routine for nonmaskable interrupt requires no special discussion here. The routine must do whatever is necessary to meet the condition warned of by the occurrence of the nonmaskable interrupt. Only one feature of this routine is different. The return to the interrupted program, if there is to be a return, is accomplished by the **RETN** instruction rather than **RET**. This is because the status of interrupt enable for normal interrupts is saved when the nonmaskable interrupt occurs. Use of **RETN** restores this status to whatever it was when the nonmaskable interrupt occurred. The use of **EI** before **RETN** is therefore superfluous. The enable state caused by the **EI** will be overridden by the action of the **RETN**.

The Z80 has yet another mode of interrupt return, namely **RETI**. This instruction functions exactly like **RET**, but generates a pulse to the interrupting peripheral signifying that the interrupt service routine has been completed. This is used in some special Zilog peripheral configurations in which the peripheral interrupts are chained into a priority configuration, which allows peripherals of high priority to interrupt the service routines of lower priority peripherals. Creation of a system which uses this scheme should be approached with great caution and used only as a last resort if nothing else will work. The reason for this is obscure, but important enough that it needs explanation here.

Imagine that there is a subroutine named **SINEX** located somewhere in memory, which computes the trigonometric sine of an angle left in HL on entry. Such a subroutine is complicated enough that the necessary arithmetic cannot be done while holding all temporary results in registers. At one or more points intermediate results must be stored in memory. Suppose now that a level 3 interrupt occurs whose service requires the computation of a trigonometric sine. The interrupt service routine loads the angle into HL and executes:

**CALL SINEX**

**SINEX** proceeds to begin the computation of the necessary sine and has stored some intermediate results, when a level 1 interrupt occurs. The level 1 interrupt service routine also requires that a sine be computed. This routine loads the angle into HL and executes:

**CALL SINEX**

which proceeds to compute the sine required by the level 1 interrupt service routine, storing temporary results all the while into the same memory locations which were being used to compute the sine required by the level 3 interrupt service. When the level 1 interrupt service routine has finished, it relinquishes control back to the level 3 routine, in the middle of its sine computation. The result is garbage for the level 3 result, because its temporary results have been destroyed by the level 1 computation.

This is a particularly nasty situation because, in general, it is never possible to recreate the exact circumstances under which the error occurred, a situation true of all interrupt related errors. The **SINEX** routine can be tested and retested with all possible input values and yield good results, but still give garbage while running because of the interference of the higher level interrupts. This situation has a name. It is known as the problem of *reentrancy*.

There are two general approaches to the solution of this problem. The simplest one is to keep a copy of the **SINEX** routine for each level of interrupt which might use it. This works, but is wasteful of memory. The second, and by far the subtler, is to write the **SINEX** routine in such a way that it will not be disturbed by being interrupt-

ed in the middle of its execution. A routine written in this way is said to be *reentrant*.

The creation of reentrant programs is no simple matter. Reentrancy puts an overhead burden on the program which will very likely cancel any advantage which might accrue to its use. In general, the working programmer who is required to produce a result will find it most productive to find ways of avoiding reentrant situations rather than solving the problems associated with them. For this reason the writer recommends that the use of this type of priority interrupt system be avoided whenever possible, and approached only with great care and caution after the programmer has built up his experience with simpler types of interrupt systems.

Returning now to the general subject of interrupt service programming, there are three general categories into which most interrupt programming errors fall. They are:

- 1) Failing to save and restore the status of the interrupted program. This causes chaos in the interrupted program, since registers appear to change value as if by magic.
- 2) Failing to reenale the interrupt system at the end of interrupt service. This assures that the interrupt service routine will be executed once and only once. This one is fairly obvious from the symptoms. A real time clock routine will increment the pulse count only once, for example.
- 3) Failing to remove the source of interrupt, i.e., the ready flag which caused it. This puts the computer into an interrupt loop from which it can never recover. This is also fairly obvious and should be suspected if the computer is stopped repeatedly and the PC always contains an address within the interrupt service routine.

In finding errors in a noninterrupt environment it is always possible to trace through the program, one instruction at a time if necessary, or by the methods to be detailed in chapter 17, until the offending instruction or sequence of instructions is located. The process can be accompanied by much misery and suffering, but it is always possible. With errors caused by faulty interrupt service, however, it

is not in general possible to recreate the situation which caused the error. Interrupts occur at points which are random with respect to the operating program. This writer has experienced a real time system which ran for 3 months, 24 hours per day, before showing the error. It did not appear again for another 6 months. Needless to say, such an error is maddening to track down. The best approach to the entire problem is to be exceedingly careful in writing interrupt service routines. Test the routines for function outside the operating program environment with tests designed in such a way that if an error occurs it can be localized easily. But mainly, be very, very careful with the original programming.

Finally, there is the matter of the interrupt instruction supplied by the peripheral interface during mode 0 interrupt operations. These instructions, known as restarts, are also available to the programmer and can be written in line in the program. Executed under these circumstances they do not cause the interrupt system to be disabled. They can be written in the assembly language using either of two synonymous mnemonics, **RST** or **TRP**. The **TRP** means trap, which is more suggestive of their real function than restart. **RST** and **TRP** require an operand in the 0 - 7 range. When the **RST** or **TRP** is executed control is sent to the address specified in the operand, multiplied by 8. Thus:

**RST     4**

sends control to memory location  $32_{10}$ , while:

**TRP     7**

sends control to location  $56_{10}$ , or  $38_{16}$ . The PC is pushed onto the stack by execution of **RST** or **TRP**, just as during interrupt operation or a **CALL**. A use for these instructions will be illustrated in chapter 17. An example of the use of in line **TRP** instructions as **CALLs** is in the bootstrap loader in appendix A.



## 17 / PROGRAM DEBUGGING TECHNIQUES

“There is no mistake;  
there has been no mistake;  
and there shall be no mistake.”

Arthur Wellesly  
Duke of Wellington

When, at long last, the processes of program writing, desk checking and assembly have been completed there remains only the process of trying out the product. Perhaps the word only implies a too casual attitude toward a process which may well be lengthier and more painful than the creation of the original program.

The final step of the assembly process is the production of the object program. Object output is produced by our assembler by means of the #O. command. When this command is issued the assembler replies with:

READY PUNCH, SPACE

The system then waits for the punch to be turned on and the space bar to be struck. The object program, a binary version of the source program suitable for loading into the Z80 system and running, is produced. The object program is preceded and followed by several inches of blank tape. When the punching is finished the system again pauses waiting for the space bar to be tapped. When the space bar is struck the prompt character (>) appears.

The loading of the object program and the supervision of its execution is performed under control of a program called the *debugging monitor* or debug for short. The full source text of this program as well as the directions for initializing it for particular configurations of peripherals are given in appendix C. If you have not already performed this initialization for your system, go to appendix A now and

do it, since you cannot proceed with the work of this chapter until this is done.

This debug program, needing a name, was dubbed ZEDBUG. The most natural name for a Z80 debug program would have been ZBUG or ZEBUG, but the literature already contains references to those names, so the program was named ZEDBUG after the European manner of pronouncing the letter Z, even at the risk of the name being confused with that of certain disagreeable creatures which inhabit cheap hotels.

ZEDBUG is entered by setting the PC of your system to its beginning location, 3600<sub>16</sub> as given in the appendix. This can easily be changed by transcribing the listing and reassembling for any desired origin. When entered, ZEDBUG immediately produces the pound sign (#) as a prompt character. This means that it is ready to accept a command. To load the object tape produced by the assembler type an L followed by a period. The transaction looks like this:

#L.

If your system has paper tape reader control the object tape will slew through the reader and stop, ZEDBUG producing another prompt character at the end. The object tape produced by the assembler contains certain validity checking data known as a *checksum*. This checksum is calculated by the assembler as the object tape is punched, and punched as the last frame of each tape record. When the object tape is read by the loader in ZEDBUG this checksum is recalculated and compared with the checksum on the tape. If the two are not identical the loader warns the user of the error by typing:

CKSM

and returning control to ZEDBUG. If this happens make sure that the tape read mechanism is clean and functional. Photoelectric readers in particular are sensitive to small specks of material in the read mechanism. After you have blown out the dust, dead insects and other debris from the reader try again. If you still get the CKSM message the tape is probably bad. Make another one and try again.

When the object program has finally been loaded, ZEDBUG will again respond with a prompt character. Its facilities may now be

used to examine the program in memory. The contents of a memory location or group of locations may be displayed on the terminal by using the M command, followed by the desired address, followed by a period. Suppose the program in example 9-5 had been assembled with an origin of :1D00 and loaded. The assembly listing for this program is shown below.

## PAGE 01      EXAMPLE 9-5

```

002                                ORG   :1D00
003                                * FILL MEMORY AREA. 1ST ADDRESS IN HL.
004                                * COUNT IN BC. FILL CHAR IN A. RETURN
005                                * WITH NO DISTURBED REGISTERS
006 1D00 F5                        FILL   PUSH   PSW
007 1D01 E5                        PUSH   H
008 1D02 C5                        PUSH   B
009 1D03 D5                        PUSH   D
010 1D04 54                        MOV    D,H
011 1D05 5D                        MOV    E,L
012 1D06 13                        INX     D
013 1D07 0B                        DCX     B
014 1D08 77                        MOV     M,A
015 1D09 EDB0                      $      MW+R
016 1D0B D1                        POP     D
017 1D0C C1                        POP     B
018 1D0D E1                        POP     H
019 1D0E F1                        POP     PSW
020 1D0F                          RES     3
021 1D12                          END

```

```

*****
* S Y M B O L   T A B L E *
*****

```

```

FILL   1D00

```

Executing the M command followed by the address 1D00 would type out the contents. This is shown below, with the parts typed by ZED-BUG underlined, and those typed by the user not underlined. The symbol @ will be used in places where the user types a blank. The examination of location 1D00 is accomplished by:

```

# M1D00.
1D00 F5

```

ZEDBUG then waits for a response from the user. If a hexadecimal number is typed the contents of the memory location just printed out will be replaced with what was typed. To change the contents of location 1D00 from the F5 to an E7, just type E7 after ZEDBUG prints the F5. The transaction now looks like this:

```
# M1D00.  
1D00 F5 E7
```

If any non-hexadecimal character is typed as replacement contents the command will be immediately aborted with a question mark and the contents of the location will be unchanged. If a mistake is made in typing, say E8 instead of the desired E7, just keep typing. Only the last two characters count. If the contents are not to be changed and no further examination is to be done, type a period. This will result in a prompt character and ZEDBUG is ready for another command. If examination of the next high location is desired, type blank and the address and contents of the next location will be typed. This may be carried out to any length. The first 5 locations of the program can be examined and/or modified by:

```
# M1D00.  
1D00 F5 @  
1D01 E5 @  
1D02 C5 @  
1D03 D5 @  
1D04 54 .  
#
```

Again, the @ signifies a blank typed by the user. The period typed in the last line terminated the M command. The blank typed after each preceding line caused the examined address to be incremented. The examined address may also be decremented by responding with a comma instead of a blank. Locations 1D04 through 1D00 are examined by:

```
# M1D04 .
1D04 54 ,
1D03 D5 ,
1D02 C5 ,
1D01 E5 ,
1D00 F5 .
#
```

The use of comma instead of blank decremented the examined address instead of incrementing it. In all cases, if a hexadecimal number had been typed in before the blank, comma or period, the contents of the memory location currently being examined would have been replaced with the typed number.

The M command has been used above to examine locations one at a time. If an entire block of memory is to be examined, it can be printed out in blocks of 16 words by using the M command with two addresses instead of one. The entire program of example 9-5 is printed out by:

```
# M1D00@1D11.
```

The response to this command is:

```
1D00 F5 E5 C5 D5 54 5D 13 0B 77 ED B0 D1 C1 E1 F1 08
1D10 FA 6E
#
```

i.e., the entire block is printed by ZEDBUG (not underlined here). A block of any length may be printed in this way. If the second address typed is less than the first ZEDBUG will reply with a question mark and ignore the command.

A similar "dumping" function may be performed by the D command. Using D instead of M punches the block of memory onto paper tape in a format which can be reloaded by the L command. This is extremely useful when patching a program in binary. It allows the corrected program to be dumped onto paper tape in reloadable



form. This permits the programmer to make a number of corrections before being forced to go back to source level to update the program, a considerable economy in debugging. The D command pauses after the period is struck to allow the tape punch to be turned on. The space bar is then struck and the punching, preceded and followed by several inches of blank tape, proceeds, followed by another pause for the space bar at the end. The spaces are not punched onto the tape.

Other memory operations performed by ZEDBUG are the fill and search functions. A block of memory may be filled with a value by the F command, followed by the first and last address to be filled, and the value. The command:

# F87@FA@EB.

fills the block which begins at :87 and ends at :FA with the value :EB. This is useful when a block must be initialized to some constant value. A memory block may be searched for some value by the S command, followed by the limits of the block and the value to be searched for. The program in example 9-5 could be searched for the value 77 by:

# S1D00@1D11@77.

the result of the command being:

1D08

#

The 1D08 being the address at which the value 77 was found. If this value had appeared more than once in the block each address at which it was found would have been logged. The S, D, or M commands may be terminated at any time by holding down the control key and pressing the C key simultaneously.

Two more commands will be considered here before proceeding to the main business of debugging programs. When testing peripheral devices it is often handy to be able to transmit single words to and from them. The I command allows input from the peripheral device.

address which follows it, the result being printed on the next line. The command:

# I21.  
B1

results in I/O address 21<sub>16</sub> being read and printed. The result, a B1 in this hypothetical case, is followed by a response from the user. If the response is a period the command terminated. If a blank is typed the operation is repeated. Data may thus be repeatedly read from a peripheral under test.

The O ("oh", not zero) command followed by an I/O address, a blank, and a hexadecimal number sends the number to the I/O address indicated. The command:

# O21 FF.

sends the number FF<sub>16</sub> to device address 21.

In all cases of ZEDBUG commands, leading zeros may be omitted. They do no harm if typed, but will have no effect. Also in all cases, an error may be corrected by simply retyping the number with the necessary number of digits. This holds for 4 digit addresses (and later for register pair contents) and for two digit memory location contents. A command will be aborted if anything but a legal response is received from the keyboard.

Having established the ground rules and examined some of the minor functions of ZEDBUG, we can now proceed to the real business of this chapter — program debugging. ZEDBUG contains a number of pseudoregisters which correspond one-for-one with the real registers of the Z80. These pseudoregisters may be displayed and modified much as memory locations have been modified in the previous examples of this chapter. All access to the pseudoregisters is via the R command. The command:

# R.

results in the following typeout:

```

      BC      DE      HL      A      S Z - A - V N C
0000 0000 0000 00  0 0  0  0 0 0
'0000 0000 0000 00  0 0  0  0 0 0
      X      Y      SP      DIFF
0080 0000 0000 0000
#

```

which displays the status of all of the ZEDBUG pseudoregisters. The above display shows the status of the pseudoregisters when ZEDBUG is initially loaded, i.e. they are all zero.

This looks more complicated than it really is, so we'll take it one step at a time. The top line of the typeout shows the status of the principal register set, BC, DE, HL, A and F. BC, DE and HL are *always* treated as pairs, while A and F are handled individually. The bits of the F register are spread out for easy viewing. The second line of the typeout is the content of the alternate register set, the one you can access after executing **EXX** and **XAF** instructions. This is indicated by the apostrophe or prime mark at the left end of the line. The bottom line shows the contents of the pseudo X, pseudo Y and pseudo stack pointer. The meaning of DIFF will be passed over for the moment. A pseudoregister or pseudoregister pair is individually examined or set by typing the R command, then the register name, then a blank. For register pairs BC, DE and HL only the name of the left-most register is used. The name F is used for the flag register. The contents of pseudo HL can be examined and modified just like a memory location. The command:

```
# RH@
```

results in:

```
# RH@0000
```

ZEDBUG then waits for a response if a period is typed nothing is changed and a prompt character appears again. If a hexadecimal value is typed, the value of the pseudoregister or pair is replaced with that value. The HL pair is set to the value 7B2F<sub>16</sub> by:

```
# RH@0000 7B2F.
```

The command must be terminated with a period or the replacement does not take place, the only result being a question mark and another prompt. The BC and DE pairs are set similarly. Remember when setting a register pair that both registers must be set. In the case of A only two digits need be typed, or one if the leftmost is a zero. Pseudo A register is set to the value 3 by:

# RA@00 3.

A leading zero before the three would have done no harm. It is unnecessary, however. If both of the above changes are made and the R. command used again, the result is:

```

      BC      DE      HL      A      S Z - A - V N C
0000 0000 7B2F 03      0 0      0      0 0 0
0000 0000 0000 00      0 0      0      0 0 0
      X      Y      SP  DIFF
0080 0000 0000 0000
#
```

The bits of the pseudo F register are set individually. The command # RF@ produces the following display:

```

S Z - A - V N C
0. 0  0  0 0 0
```

ZEDBUG then pauses waiting for a keyboard response. If the name of one of the flag bits is typed, followed by a 1 or a 0, the flag value is replaced by the typed value, ZEDBUG types one space, and waits for the name of another flag. This may be continued until all the flags have been set to their desired values. The setting of the S, A and C flags is accomplished by:

```

# RF@
S Z - A - V N C
0 0  0  0 0 0 S1 A1 C1.
```

The flag settings need not be done in left to right order as shown here. After the above has been typed in, another # RF@ command will yield:

```
S Z - A - V N C
1 0 1 0 0 1
```

The alternate pseudoregisters (i.e., the prime registers) are set and manipulated exactly like the principal registers except than an apostrophe is typed after the register name instead of the blank (@ in our displays here). The command:

```
# RB'
```

results in:

```
# RB'@0000 3F8.
```

with the value of BC' being set to 03F8<sub>16</sub>. With this change an R. command yields:

```
BC    DE    HL    A      S Z - A - V N C
0000 0000 7B2F 03      1 0 1 0 0 1
03F8 0000 0000 00      0 0 0 0 0 0
X      Y      SP  DIFF
0080 0000 0000 0000
```

showing the cumulative changes which have been made. The X, Y and SP pseudoregisters are set in exactly the same way as the other pseudoregister pairs, SP being addressed as S.

The best way to become familiar with all of these mechanics is to exercise your knowledge. At this point it will be well worth your while to abandon reading for a while and try out the things which have been discussed so far in this chapter. When you are comfortable with the loading process, the examination and inspection of memory and the setting of the pseudoregisters come back here and we will proceed.



Example 17-1 shows the assembly listing of a program which is to convert the fraction in the A register to ASCII-decimal, leaving the result, beginning with a decimal point, in the five word area whose lowest address is contained in X on entry. The example contains deliberate mistakes which we will find and correct using ZEDBUG. The object tape is loaded into memory using the L command, and the presence of the program verified by printing it out with the M command, i.e.:

```
#M1E00 1E2A.
1F00 DD 36 00 AE DD 23 06 04 6F 26 00 29 54 5D 29 29
1E10 09 3E B0 B4 DD 77 00 DD 23 10 FE C9 DD 21 26 1F
1E20 CD 00 1F BF EB E9 DF BD F1 95 FB
```

Do not omit this precaution. Always check that the program has been loaded, at least in a few places if not the entire program. This can save you from some mysterious situations.

The program will be executed under the control of ZEDBUG by means of a procedure called *breakpointing*. Breakpointing is the most powerful single debugging tool you have, at least in conventional programming systems, so learn to use it intelligently. It is not without its drawbacks, but it offers a faster route to the finished, working program than any other procedure. The effort you put into understanding breakpointing will repay itself very quickly.

### Example 17-1

```
PAGE 01      TEST OF FRACTION CONVERTER

002          ORG      1E00
003 1E00 DD3600AE $ CSF    SIX 0.
004 1E04 DD23 $          INX X      INCREMENT POINTER
005 1E06 0604          MVI B,4     DIGIT COUNT TO 5
006 1E08 6F          MOV L,A      MOVE FRACTION TO L
007 1E09 2600          MVI H,0     CLEAR INTEGER SPACE
008 1E0B 29          DAD H        MULTIPLY BY TWO
009 1E0C 54          MOV D,H      SAVE IN DE
010 1E0D 5D          MOV E,L
011 1E0E 29          DAD H        TIMES FOUR
012 1E0F 29          DAD H        TIMES EIGHT
013 1E10 09          DAD B        ADD PREVIOUS RESULT
014 1E11 3E00          MVI A,'0'   GET "FRAME" FOR ASCII DIGIT
015 1E13 84          ORA H        MERGE WITH BCD DIGIT
016 1E14 DD7700 $     SAX 0       STORE IN STRING
017 1E17 DD23 $       INX X      INCREMENT STORE POINTER
```

```

018 1E19 10EE $ DBNZ CSF1 DECREMENT COUNT
019 1E1B C9 RET DONE WHEN COUNT ZERO
020
021 TEST CODE FOR CSF
022
023 1E1C DD21261E $ TCSF LXI X,N
024 1E20 CD001E CALL CSF
025 1E23 RES 3 SPACE FOR BREAKPOINT
026 1E26 N RES 5
027 1E2B END

*****
* S Y M B O L T A B L E *
*****

CSF 1E00 CSF1 1E09 N 1E26 TCSF 1E1C

```

Breakpointing is a controlled execution of part of the program being tested. When a breakpoint is executed, three words of the program being tested are replaced by a **JMP** instruction, which will return control to ZEDBUG when that point is reached. Looking at the program of example 17-1, the test code begins at location :1E1C with the loading of the X register and ends at :1E23. The **RES 3** at this location is simply allowance for the **JMP** instruction which will be inserted by ZEDBUG. We want to begin execution at address :1E1C and return to ZEDBUG when control reaches :1E23. This can be accomplished by the command:

# B1E1C@1E23.

But don't be too hasty and try this prematurely. There is the question of how to get the number which is to be converted into A. The answer is quite simple. Those pseudoregisters we spent so much time on a little while ago are loaded into the real registers before ZEDBUG executes the jump to :1E1C. All machine registers are loaded from their pseudoregister counterparts before the jump, so that the real register status as the program begins execution is exactly that which was specified by the pseudoregisters in ZEDBUG. We therefore set the pseudo A register to the required test value. For testing a fraction converter the value  $.75_{10}$  is a good place to start, so we type:

# RA@03 C0.

This replaces the former value of pseudo A, a 3, with the value C0<sub>16</sub> which is .75 with the point all the way to the left. We now try the breakpoint by typing:

```
# B1E1C@1E23.
```

and nothing happens. Nothing at all. When the machine is stopped the program counter is nowhere near the program being tested. It may even be in nonexistent memory. The first thing to do is set the PC back to the beginning of ZEDBUG and start the computer. This will restore the three program words which were replaced by the breakpoint and get the program back to its original condition. Maybe. Because the program ran out of control there is no way to know what damage may have been done by the random execution of instructions which took place. If ZEDBUG does not respond when it is restarted, then ZEDBUG itself has been destroyed. It will be necessary to reload both it and the program to be tested. If ZEDBUG responds, count yourself lucky. This dumb error did not do any serious damage. Let's figure out what the error was.

Recall that all machine registers are loaded from their pseudo counterparts before the **JMP** from ZEDBUG to the beginning execution address, :1E1C in this case. This means *all* registers, including the stack pointer. Not having bothered to set the pseudo stack pointer, it was zero at the time of the jump. Since the second executed instruction is a **CALL**, this pushed the PC into imaginary memory (on a system with less than 64K RAM). The subroutine may or may not have executed in this case, but when the **RET** at 1E1B was encountered, nonsense was popped into the PC from this imaginary memory space. The program counter then departed for some unpredictable place, "Gumbiland", as Mr. Frank Horne is fond of calling it. We have committed the commonest error in all microcomputer programming, forgetting to load the stack pointer.

We could at this point go back to the source program and insert the necessary **LXI SP** instruction, but ZEDBUG offers an easier way out. Set the pseudo stack pointer to some acceptable value so we can try again. Suppose the space below :100 were not in use. The stack can be set to this value by:

```
# RS@0000 100.
```

After again setting pseudo A to C0 (necessary if ZEDBUG had to be reloaded) we try the breakpoint again.

# B1E1C@1E23.

and the ZEDBUG monitor replies immediately with:

```

BKPT REACHED AT 1E23
  BC   DE   HL   A       S Z - A - V N C
0000 0000 0100  B1     1 0   0   1 0 0
03F8 0000 0000  00     0 0   0   0 0 0
  X    Y    SP  DIFF
1E2B 0000 0100 0000

```

At least this time something happened. Exactly what happened is this. A **JMP** instruction leading back to ZEDBUG was placed in :1E23 and the following two words, the former contents of these words being saved within ZEDBUG. The Z80's registers were then loaded from the pseudoregisters and a **JMP** :1E1C from ZEDBUG was executed. The program then executed. When control reached the **JMP** which ZEDBUG had placed at :1E23 and the two following words, the **JMP** was executed. Upon rearrival in ZEDBUG the machine registers are stored into their pseudo counterparts and the three words beginning at :1E23 are replaced with their original pre-breakpoint values. Everything looks fine up to here.

There remains only the question of whether the thing produced the right answer. The correct conversion of :C0 should be .7500, since the routine cuts off conversion after 4 decimal digits have been generated. The ASCII for .7500 should be AE B7 B5 B0 B0, the AE being the ASCII for the decimal point. We test this by printing out the result area beginning at N, location :1E26.

```

#M1E26 1E2A.
1E26 AE BA B3 B2 B1

```

Garbage. The converter doesn't work. Outside of the AE for the point, not a single character is correct.

To find the problem we will have to step through the routine in smaller increments, checking for correct results at each stage. We begin by resetting pseudo A to C0, as before and executing only the portion of the program from 1E1C to 1E09, i.e., the **LXI**, the **CALL** and the first four instructions of the subroutine.

```
#RA B1 C0.
#B1E1C 1E09.
BKPT REACHED AT 1E09
  BC   DE   HL   A       S Z - A - V N C
  0400 0000 01C0  C0     1 0   0   1 0 0
'03F8 0000 0000  00     0 0   0   0 0 0
  X     Y     SP  DIFF
  1E27 0000 00FE FFFE
```

Now look at this partial result. The digit count, a 4, is properly placed in B. A has been copied to L and the stack pointer now reads FE instead of 100. This difference in the stack pointer was caused by the pushing of the PC during the **CALL CSF**. Note the DIFF column. It displays the number FFFE. This is the difference between the stack pointer values when the breakpoint was entered and when control returned to ZEDBUG. This is of no particular significance here but this information can be of great value when debugging programs which make strenuous use of the stack. It is a means of telling at any time whether the proper number of pushes and pops has been executed. Now let's execute the next instruction. To do this it will only be necessary to type one address. If only one address is given, the breakpoint facility will use the return point from the previous breakpoint as its jump address. If there was no previous breakpoint ZEDBUG will give you a nasty message. To resume execution at 1E09 and stop at 1E0B, we need only type:

```
# B1E0B.
```

and the result is:



```

BKPT REACHED AT 1E0B
  BC   DE   HL   A       S Z - A - V N C
  0400 0000 00C0  C0     1 0   0   1 0 0
  '03F8 0000 0000  00     0 0   0   0 0 0
    X    Y    SP  DIFF
  1E27 0000 00FE 0000

```

To this point, everything looks proper. The H register has been cleared to zero. Now let's try the first left shift and copy to DE.

```

#B1E0E.
BKPT REACHED AT 1E0E
  BC   DE   HL   A       S Z - A - V N C
  0400 0180 0180  C0     1 0   0   1 0 0
  '03F8 0000 0000  00     0 0   0   0 0 0
    X    Y    SP  DIFF
  1E27 0000 00FE 0000

```

Again, this looks OK. The C0 in HL has been properly shifted left to get 0180 and this has been copied into DE. In case it hasn't become obvious by now, the breakpoint is executed up to but not including the named address, i.e., the last instruction executed in this breakpoint was the **MOV E,L** at 1E0D. The **DAD H** at 1E0E was not executed. The second address names the location at which the return should occur. So far so good, now we'll try the next two left shifts.

```

#B1E10.
BKPT REACHED AT 1E10
  BC   DE   HL   A       S Z - A - V N C
  0400 0180 0600  C0     1 0   0   1 0 0
  '03F8 0000 0000  00     0 0   0   0 0 0
    X    Y    SP  DIFF
  1E27 0000 00FE 0000

```

Again the result looks correct. The 6 in the H register is the correct result of shifting 00C0 in HL left three places, and the copy of the doubled original number is still in DE. Now we execute one more instruction.

```
#B1E11.
BKPT REACHED AT 1E11
  BC   DE   HL   A       S Z - A - V N C
  0400 0180 0A00  C0     1 0   0   1 0 0
'03F8 0000 0000  00     0 0   0   0 0 0
  X     Y     SP  DIFF
  1E27 0000 00FE 0000
```

and it's all wrong. Since the first digit generated had to be a 7 if the value of the original fraction was .75, the A which has appeared in H is wrong. Looking at the assembly listing quickly reveals the problem. We are adding the wrong register pair to HL. The result of the first left shift was saved in DE, but we are adding BC. This kind of error is common enough when transcribing the written program. The question is how to fix it.

The most obvious way is to go back to the source level and correct the original program. This works, of course, but takes some time, particularly if the listing device is slow. This has to be done eventually, but it would be better to find some way to patch the error temporarily so we can proceed with testing the program. If any more errors are found they can all be fixed at once in the source program and only one listing pass made instead of several.

With the aid of the M command, the offending instruction can be changed. From appendix D we find that the hexadecimal configuration of a **DAD D** instruction is 19. The patch is made by:

```
# M1E10.
1E10 09 19.
```

The A register is now reset to its original value, C0, and the program run again:

```
#RA C0 .
#B1E1C 1E23.
BKPT REACHED AT 1E23
  BC   DE   HL   A       S Z - A - V N C
  0000 0000 0000  B0     1 0   0   0 0 0
'03F8 0000 0000  00     0 0   0   0 0 0
  X     Y     SP  DIFF
  1E2B 0000 00FE 0000
```

The result is then examined using the M command:

```
# M1E26 1E2A.  
1E26 AE B7 B5 B0 B0  
#
```

and found to be correct, .7500. With this result correct, we can breakpoint through the program with other test values to verify its correct performance under all circumstances. Note that the stack pointer reads 00FE in this last breakpoint report. It would be of value for the reader to trace back through what has been done to find out why this is so.

We cannot leave this discussion of breakpointing without pointing out some cautions that must be observed. First, it must be obvious by now that both of the breakpoint addresses must be the locations of the first word of an instruction. If this is not so either the **JMP** from ZEDBUG will land in the middle of the instruction or the breakpoint return will not be executed. If control does not return to ZEDBUG, from this circumstance or any other, stop the computer, set the PC to the beginning of ZEDBUG and start it again. This will insure that the three words replaced by the breakpoint return **JMP** will be restored.

Note also that the breakpoint return instruction requires three words. This means that breakpoints must be carefully placed. If a breakpoint had been attempted from the **LXI X,N** at 1E1C to the **RET** at 1E1B the last two words of the return **JMP** would have overlaid the first two words of the **LXI**, leading to disaster.

The error patching case covered above is a simple one, the replacement of a one word instruction with another one word instruction. If an instruction needs to be removed, the fix is even simpler. Just use the M command to set the offending instruction to zeros. Make sure you set *all* of it to zero, if it is a multiword instruction. The case of the missing instruction is a bit harder.

If the patch requires that an instruction of one or two words be replaced by one of two or three words, or if an instruction has been omitted, another procedure is used. The point at which the instruction must be inserted is replaced with one of the restart instructions to which no interrupt is attached. These instructions are:

|    |             |
|----|-------------|
| C7 | traps to 0  |
| CF | traps to 8  |
| D7 | traps to 16 |
| DF | traps to 24 |
| E7 | traps to 32 |
| EF | traps to 40 |
| F7 | traps to 48 |
| FF | traps to 56 |

If the instruction replaced by the restart is more than one word long set the remaining words to zero. At the trap location the effect of the replaced instruction must be duplicated, followed by the instruction which is to be inserted. The last instruction of the patch must be a :C9, which is a **RET**.

There are three last features of ZEDBUG which must be covered, namely the J and Slash (/) commands and the stack examination feature. J means a simple jump after loading the Z80's registers with the contents of the pseudoregisters. It is something like a breakpoint, except that control does not return to ZEDBUG. The J accomplishes a simple transfer of control to the program to be executed. Needless to say, it is only used when a good deal of confidence has been built up in the program to be run. J accepts only a single address and terminates with a period.

The slash command controls the enablement of interrupts during breakpoints and jumps. The command:

# /D.

will result in the interrupt system being disabled before a breakpoint or jump, while:

# /E.

results in breakpoints and jumps being executed with interrupts enabled.

The contents of the stack may be examined directly by using the M command followed by the letter S and a period. This command fetches and prints the word pointed to by the pseudo stack pointer,

and the word at the next higher address, i.e., the last item pushed onto the stack by the user program. This is useful when breakpointing into a subroutine since it displays the address of the calling point. It is further useful at the end of the subroutine to insure that the correct return address has been placed in the right stack position. Typing space displays the item previously pushed, i.e., displays the item one deeper in the stack (toward higher memory). Typing a comma steps in the other direction. The command is terminated by typing a period. MS always displays two memory words, the lower memory address being the low order half. The pseudo stack pointer is not changed.



## 18 / FUNCTIONAL DESCRIPTION OF THE ASSEMBLY PROGRAM

“It went ‘zip’ when it moved, and ‘bop’ when it stopped,  
And ‘whirr’ when it stood still  
I never knew just what it was and I guess I never will.”

Chorus to *The Marvelous Toy*  
Song by Tom Paxton.

Anyone who has read this far in a book like this must be viewed as a serious assembly language programmer, at least in prospect. Following this calling involves, sooner or later, finding out something about the assembly process itself. It is to this need that the present chapter is addressed.

The assembler in appendix B was specifically created to support the language used in this book. Further, to make it useful to the greatest possible number of readers, it is written in the 8080 subset, and is completely input/output independent, so that it can be run on any 8080 or Z80 based system with sufficient memory. The assembler itself requires about 10K of RAM, the exact amount depending on the length of the I/O drivers supplied by the user. It contains no self modifying procedures other than the filling in of character strings in message frames, and can therefore be put into ROM with a minimum of modification.

It was created with the secondary function of providing an instruction vehicle for those who might find it necessary to delve into the mechanics of the assembly process itself. It is, therefore, not intended to be a software tour de force or an example of optimal assembler design, but rather the most easily understood program which could be written to serve the purpose. To this end only the simplest possible techniques have been used. Instruction and symbol tables are separate and accessed by simple table lookup techniques. The sorting of

the symbol table is done by a simple bubble sort. Exotic technique has been completely excluded for the sake of clarity, even at the sacrifice of efficiency.

Another constraint on the assembler was that it should be written for the convenience of the working programmer, not that of the assembler writer. While it may save some coding labor and simplify the design of the assembler to prohibit the working programmer from using labels which coincide with the names of machine registers, it is preposterous to place this burden on the eventual user for the simple convenience of the assembler writer. The assembler writer need only grapple with the problem once when the system is created. Hundreds or thousands of users must live with the senseless restrictions afterwards. This outlook has therefore been firmly rejected.

The text editor has been similarly designed. It is the simplest possible line oriented editor which will do the job. No exotic data structures have been employed. The information is pushed about by the editor in what some might call "brute force" fashion. This again results in some loss of efficiency when the program becomes long. When adding a line, for example, at the beginning of a program 1000 lines long there is a perceptible pause of a fraction of a second before the addition is acknowledged, particularly on a system running at 2 Mhz. Such is the cost of simplicity. Both text editor and assembler have been subjected to extreme testing and contain no known deviations from the specification of this book. If any are discovered by readers, the writer would greatly appreciate knowing of them so that they may be corrected in later printings.

In the discussion of this chapter, reference will be made to various symbols. To find the locations corresponding to these symbols look them up in the symbol table at the end of the assembly listing. Execution of the editor/assembler, known hereafter collectively as the assembler, begins at location **MON**, just below the Frankenstein quote on the first page of the assembly listing. As has been pointed out in previous chapters, forgetting to load the stack pointer is a common error, so we do it first. The space immediately below **MON** is used as the stack space. The four **CALLs** which follow initialize the interfaces for the terminal device, paper tape reader, paper tape punch and listing device, all routines supplied by the user for his own configuration, as described in appendix B. The reference to **FLAG** which follows the calls determines whether the pointer to

the source program work space, known as the *source buffer*, has been initialized. If it has not control goes to **CLR2** which performs the task. The flag is then cleared so that this branch of the program is not taken again. This procedure is executed only once when the program is loaded cold. The **CALL** at location **MON1** is for the initialization of a general I/O device, a stub for future expansion to disk. The stack pointer load at **MON2** seems redundant, but is required by the return to this point from certain procedures which leave the stack pointer disturbed.

The main monitor loop of the assembler is then entered, with the terminal carriage being returned and the prompt character (>) being typed. The program then waits for keyboard input (RKB). The character typed is then compared to the commands in the legal command list, i.e., /, L, D, A, R, P, and #.

The / command, used when the buffer is to be cleared in preparation for work on a new program, reinitializes the source buffer pointer and location **NXTCHR** to the first available buffer location. **NXTCHR** is used during the editing process to indicate the next available source buffer word. As noted before, the initialization flag is also cleared.

The L command is a bit more complicated. It uses the same program machinery as the P (punch) command. This intertwining of code dates from the origin of the assembler when only a Teletype was available as both listing and punch device. The list function begins at location **LIST** with the clearing of a flag which differentiates between the list and punch functions. The **CALL FPRM** fetches a pair of line numbers from the terminal keyboard, tests them for legality, i.e., do these lines exist and is the second line number greater than the first, and returns the memory addresses of the beginning character of each of the lines. The address of the beginning character of the line is found by counting the carriage returns which terminate each line in memory. The addresses of the first and last characters of the lines are left at **BADR1**, **EADR1**, **BADR2** and **EADR2** respectively. **FPRM** in turn calls **PFCH** (parameter fetch) to get the individual line numbers from the keyboard. It is at the **PFCH** level that the detection of the F symbol (for final line) is done. The line number is then converted and printed, followed by the text of the line which begins at **BADR1**. When a carriage return is encountered the process is repeated for the next line, this continuing under control of a count

(**IDIFF**) which is the *inclusive* difference of the two line numbers named in the keyboard request. Note that the inclusive difference is used. This is because a request like

**L10 15.**

implies the printing of six lines, not five. When the line count is exhausted control returns to **MON1**.

The **D** command gets its limit parameters from the keyboard using **FPRM**. The inclusive difference **IDIFF** is again used to determine the number of lines to be deleted. The difference between the current line count **LNCNT** and **IDIFF** forms the new line count. The first character of the first nondeleted line above the deleted portion is then moved into the space occupied by the deleted material. Using the difference between this first nondeleted character address (**EADR2+1**) and **NXTCHR** the entire upper buffer is then moved down by the number of words indicated by **EADR2-BADR1**.

The **A** function is by far the most complicated in the editor. A single line number is fetched by **PFCH**, the address of the beginning and end being determined by **FBA** (find buffer address). The character position one beyond the end of the named line will be the first word of the line to be added. This is the reason for the **INX D** just above **ADD0**. The carriage is then returned, a pointer to a buffer area at **KB** set into **HL** and the system waits for the typed input line. When the carriage return is encountered, the length of the typed line is determined and the entire source buffer above the point of insertion moved up by the amount required to accommodate the new line, if the existing buffer space will allow it. If insufficient space exists a buffer full message is delivered and the command is aborted. When the move has been completed, the new line is copied into the space thus created. **NXTCHR** is increased by the length of the line added and **LNCNT** is incremented.

The **R** command reads text from the system paper tape device and appends it to the end of whatever is already in the source buffer. With each line read, **NXTCHR** and **LNCNT** are updated.

The **P** command follows the same general track as **L**. The differences are that pauses are executed at the beginning and end of punching, waiting for the space bar to be pressed. This is to allow for the situation in which a Teletype or similar device has the com-



mon function of punch and listing device. The pauses (**CALL SB**) can be deleted if desired, along with the printing of the **READY PUNCH, SPACE** message.

The **#** command sends control to the assembler proper, location **ASM**. At this point we must pause in the description of the mechanics to describe some logical characteristics of assemblers. What must be understood about this process from the outset is that the assembly program sees the user source program *one line at a time*. A single line is read into an internal assembler buffer from the source program buffer or from paper tape. This line is then processed as a single entity. This structure, viewing the program a single line at a time, requires that the assembler process the source program twice. The first pass has as its principal function the formation of the *symbol table*, the list of all symbols which have been used as labels in instructions or pseudo-ops and their values. The second assembler pass produces the actual object program using the symbol table to fill in the values required by the operands of the source statements.

The reading of the source record into the assembler's internal buffer is not a simple transcription of the information from one place to another. Since the assembler allows freeform input, one space indicating an entire blank field, the record must be reformatted internally to be processed. Note that even when the source text is only being listed, not assembled, it appears in rectified form, with all component fields in their proper places, however the source was originally typed in. This transformation is done by the **FORMAT** routine. The first character of the source record is examined for blank. If it is not blank the first six are copied into the internal buffer at **LABEL**, or characters are copied until a blank is encountered. If more than six characters appear in this label the remainder are ignored. The scanning then continues until more nonblank material is reached. This is copied into the instruction field of the internal buffer at **MNEM**. At this point a decision must be made about the remainder of the nonblank information in the line. If the instruction is of the **LDA, LXI, SDY** etc. variety it will require an operand. In this case the next nonblank information must go to the operand field at **OPNDF**. If the instruction is of the **RAL, NOP** or **EI** variety the remaining nonblank material must be transcribed into the comment field at **CFLD**. To determine which situation is being dealt with the instruction is looked up in the main instruction table at **ITBL**. This table lookup is



performed by the subroutine at **OPSRC**. Execution of **OPSRC** yields two results, the binary skeleton of the instruction and an instruction class word. The binary skeleton will not concern us yet. The instruction class word contains the class, indicating the type of assembly processing required for this instruction, in the lower bits. The high bit of this word is a formatting flag. If this flag is a zero the instruction requires an operand. If the flag is a one there is no required operand and the remaining nonblank text of the source line is copied into the comment field of the internal buffer. If an operand is required, the nonblank text is copied into the operand field at **OPNDF** until a blank is encountered, any nonblank material after this becoming commentary. This holds true under all conditions but one, namely that in which an apostrophe has been encountered in the transmitted string. This indicates the presence of an ASCII operand which may contain blanks. In this case the transmission continues until the first blank following the second apostrophe. In all cases of operand field transmission, if the operand length exceeds the allotted space (9 characters) the beginning of the comment field is moved to the right to make space for it. This will occur with expressions like

**LXI H, LONG-LONGER**

With all parts of the source statement in their proper fields in the internal buffer, the analysis of the line can proceed. Vital to this process is the **FANLYZ** (field analyze) routine. It scans a string of characters whose beginning address is given in **DE**, determines the field type (ASCII, decimal, hexadecimal, octal or symbolic) and returns the value of the field or sum or difference of fields to the calling program. Symbolic operands are looked up in the symbol table for their values.

Under control of the assembly monitor beginning at **ASM**, **PASS1** of the assembler is executed. The symbol table count **NSYMS** is cleared as well as locations **ILC**, **ORGVAL**, **TLFLAG** and **P2LNO**. These last two can be skipped over for now, as they perform relatively minor functions. **ILC** is the *instruction location counter*, around which the activities of **PASS1** revolve. **ILC** is used as a pointer to the address of the next word to be assembled. When a label is encountered in a source instruction it is the value of **ILC** which becomes the value of the label in most cases. **ORGVAL** is the value of the operand field of whatever **ORG** pseudo-op is in force currently. This will become clearer a little later.

The processing proper begins at **P1B** with the fetching of a source record (**FSRCD**). Though no listing is done in the first pass the line number must be converted to decimal in case an error must be flagged (**CNVLNO**). Note that the high digit at the line number is converted but discarded. This is done to allow a wider comment field in the assembly listing. The label is examined for blank or asterisk, blank meaning no label, and asterisk in the first column meaning a comment line. Comments are ignored in pass 1. Next a check is made to see if the mnemonic field contains an **EQU**, the **EQU** requiring special processing during pass 1. If it is an **EQU** the label is first checked for validity (**CALL SLLBL**), and the operand field is analyzed. The operand field of an **EQU** must have already been defined when the **EQU** appears. This error must be detected and flagged in pass 1. If the operand field consists totally of defined elements, then the label of the **EQU** is entered into the symbol table with the operand value as the symbol value.

Given that the mnemonic field does not contain **EQU**, the value of the label for this line becomes the current value of the **ILC**, if the label is a legal one. A check is made to see whether this is the first appearance of this label. If it is not, the symbol is multiply defined. This is noted in the symbol table by ANDing off the high bit of the first character of the symbol. This has no effect on the way in which this character is printed, since ASCII is indifferent to the value of the high bit. It serves as a flag for printing the M (multiply defined) error message in pass 2.

The instruction class is then used to look up the length of this instruction in the table beginning at **P1LTBL**. In a few cases in which the instruction length is ambiguous from the mnemonic a further check is made (**CALL CKZ80**) and the instruction class adjusted. The instruction length is then added to **ILC** to form its new value. If the instruction itself was undefined, i.e., not a legal Z80 instruction the length defaults to 4, the maximum length of a Z80 instruction. With **ILC** incremented by the length of the instruction the process is repeated until an **END** is encountered.

Some mnemonics require special action during pass 1. **ORG** resets **ILC** to the value of its operand field. **RES**, **BSS**, **DATA** and **ASC** may be of variable length. In these cases the field length is determined and added to **ILC** as the length of the "instruction".

When the **END** is encountered control returns to the assembly monitor. If the assembly involves listing (**#L.**) the symbol table is sorted (**CALL SRTST**) and various counters and flags are initialized for pass 2. A fresh page is then spaced up on the listing device (**CALL EJECT**) and the page number is printed (**CALL PGN8**). **CALL PASS 2** is then executed.

The actual production of the object code takes place in pass 2, using the symbol table produced by pass 1. The processing of each source record begins with the clearing of error flags and conversion of the line number. This conversion takes place even on nonprinting assemblies because the line number will be required in the event of an error print out. Remember, lines containing errors are printed out in *all* assembly modes. The label is checked for legality and multiplicity of definition, these error flags being set if necessary. The instruction class is then used to compute the location of an address in the pointer table at **P2JTBL**. This address is then picked up from the table and transferred to the PC, sending control to the appropriate instruction processor.

Each processor performs the necessary analysis to process the particular instruction or pseudo-op class it services. The binary skeleton of each instruction is used, with operand modifiers where appropriate, to form the final object instruction. This involves 25 separate processing classes, making the pass 2 processor easily the most complex part of the assembler. Each of these separate processors takes the appropriate action and sets error flags where necessary. Each determines the correct length for the instruction being processed and adds it to **ILC**, just as was done in pass 1. An example of the type of error detection done can be seen just below location **P2C2** (pass 2, instruction class 2). The **CPI @66** checks against the forbidden configuration **MOV M,M**. If this were allowed to slip through the result in the object program would be :76, not a **MOV** at all, but one of those deadly halts from which no recovery can be made if the interrupts are disabled.

Instruction classes B or greater are reserved for Z80 only instructions. Each of the processors for classes B or greater finishes its work by transferring control to location **FLGZ80**, whose purpose it is to place the dollar (\$) in the listing to the left of the label fields of Z80 only instructions. Nonexistent instructions are fed to **P2BDOP**

which defaults to an instruction length of 4 and produces 4 words of zero in the object program.

All instruction processors eventually lead to location **P2C1B**, which calls **PCHOBJ** (punch object). **PCHOBJ** does not actually punch the object code but merely collects it in a 128 word buffer, either until the buffer is full, or an **ORG**, **RES** or **BSS** is encountered. When one of these conditions arises the object buffer is punched out and a new record started. Control then goes to **P2PRNT**, the print supervisor.

**P2PRNT** converts **ILC** for listing, the line number having been previously converted. The object code, up to four words, is then converted and placed in the listing buffer which begins at **HEX**. This is all done whether or not a listing is to be produced, since the information will be required if the line must be printed because of an error. The error flag word **ERRORS** is then examined and the appropriate ASCII error indicators set in the block beginning at **ERLNE**. If there are errors, the line is printed on the terminal device unconditionally. There is no way to suppress this printing. If there are no errors and the pass does not involve listing (**#E.** or **#O.**) the location counter **ILC** is updated from the length of the instruction and the process repeated until the **END** is encountered. In the case of a **#Z.** assembly, the scan for Z80 only instructions, the instruction is printed unconditionally if it is of class B or greater. The detection of this condition is done in the **FLGZ80** routine.

Following the detection of the **END** source line control returns to the assembly monitor which prints the previously sorted symbol table and restores paper if listing was in force.

The object program is punched onto paper tape in discrete records which are of two types. The first, type 0, consists of data generated by instruction and data generating pseudo-ops like **DATA**, **ASC**, **DBL**, etc. The records consists of a record mark (:FF) a record origin, a one frame record length, and the data record itself, followed by a simple additive checksum. Type 0 records are accumulated in the buffer beginning at **RCDBLK**, being punched out as described above when the next entry would overflow the 128 word buffer, or on command from **DPOBJR** (dump object record). The occurrence of an **ORG**, **RES** or **BSS** also forces the object buffer to be punched. With **ORG** this is because the presence of the **ORG** implies that another record with a different origin, possibly quite distant in memo-



ry, will follow. With **RES** and **BSS** this is done to "close" the data record so that another record type, type 1, can be punched. **RES** and **BSS** have their own special record type because to attempt to punch them as type 0 records would lead to unacceptably long object tapes. Consider the implications of the source statement: **RES 20000**. Each of the 20,000 frames would have to be separately punched in a type zero record, divided into 128 word blocks.

A type 1 records consists of record mark :FF, the record origin, i.e., the value of **ILC** at the time the **RES** or **BSS** was encountered, a frame containing the record type, a "fill" flag, i.e., a flag set to indicate the presence of the second operand with which the reserved space is to be filled, the fill value itself, and a checksum.

The loader in ZEDBUG reads these records and loads them into the indicated places in memory. During the loading of type 1 records the system paper tape reader is stopped. This is done because a fill of excessive length could cause the reader to miss a character while the indicated fill value was being copied into the required memory space. On systems using only a Teletype as the tape read device there will be no damage done by the absence of reader control here, since the device is slow enough that no tape frame will be missed, even with large blocks.

Note that ZEDBUG is also capable of producing records in object format. The tapes produced by the ZEDBUG D command, however, contain records which are exclusively of type 0. These records may be 256 words in length, as contrasted with 128 for the object output of the assembler. Both types of tapes are loadable with the ZEDBUG L command.

What follows this chapter are 4 appendices containing cold start instructions, the source text of the assembler and the instructions for initializing it to particular I/O configurations, the source of the ZEDBUG debugging monitor with similar initialization instructions, and a cross index to allow translation between the Zilog mnemonics and those used in this book.



## APPENDIX A / LOADING PROCEDURES FOR OBJECT TAPES

Paper tape object programs for both the assembler and the ZEDBUG monitor are sent at no charge to the original purchaser of this book when the coupon at the back of this book is returned. This appendix details the procedures necessary for loading these object programs.

The object tapes supplied are in a special format which contains only type 0 records. These tapes are loaded by means of the special loader given below. The general procedure is to load the ZEDBUG object tape first, initialize ZEDBUG input/output routines, then use the ZEDBUG dump facility to dump the initialized version of itself. The ZEDBUG loader may then be used to load the assembler. When the assembler has been loaded it is initialized and the initialized version is then dumped via the ZEDBUG dump facility.

To load the supplied object tapes, the following program must be placed into memory. This may be done using an existing ROM monitor, a front panel key in procedure or whatever means is at your disposal. The program is:

| <i>location</i> | <i>contents</i> |
|-----------------|-----------------|
| 80              | 31              |
| 81              | 80              |
| 82              | 00              |
| 83              | C7              |
| 84              | FF              |
| 85              | FE              |
| 86              | FF              |
| 87              | 20              |
| 88              | FB              |
| 89              | 57              |

|    |    |
|----|----|
| 8A | FF |
| 8B | 6F |
| 8C | FF |
| 8D | 67 |
| 8E | FF |
| 8F | B7 |
| 90 | 20 |
| 91 | 0C |
| 92 | FF |
| 93 | 47 |
| 94 | FF |
| 95 | 77 |
| 96 | 23 |
| 97 | 10 |
| 98 | FB |
| 99 | 5A |
| 9A | FF |
| 9B | AB |
| 9C | 28 |
| 9D | E6 |
| 9E | 18 |
| 9F | FE |

This loader uses **TRP** (trap or restart) instructions as one word **CALLs** to shorten the amount which must be entered. In addition to the loader shown above, the user must supply two subroutines. The first, beginning at location 0 and called by the **TRP 0** at location :83, must do whatever is necessary to initialize the paper tape reader interface and start the reader in your system. When control returns to location :84 of the loader program steps must have already been taken to move tape through the reader. If the reader is as simple as a Teletype without reader control this routine may consist only of the single word :C9 (**RET**) at location 0. However long or short this routine is, it must end with :C9. There are  $56_{10}$  words available for this function, from 0 through :37. This routine must not disturb the stack pointer, but any other registers may be used.

The second routine begins at :38 and is called by the various **TRP 7** instructions in the loader. Its function is to read one frame

from the tape. It may use only the A register, disturbing *nothing* else, and it *must* end with the following sequence of words:

F5  
82  
57  
F1  
C9

which in source form is:

**PUSH   PSW**  
**ADD     D**  
**MOV     D,A**  
**POP     PSW**  
**RET**

None of the above is location dependent. It will be the same however long the read sequence for your system is. Your read sequence *must* end with these 5 words. They are the checksum computation for the loader and the load will be aborted after the first record of the tape if the checksum is incorrectly computed.

As a guide to this initial key in process, the bootstrap loader preceded by the reader start and tape read routines for the writer's system is shown in assembled form immediately following these instructions.

After all of this is in the lower memory locations indicated, set the PC to :80. Take the ZEDBUG object tape and put it in the reader. Place the tape so that the read pins or photodiodes are on the blank leader. The first punched frame of the tape should be a rubout (all holes punched). If it is not you have the tape backwards in the reader. When this is done start the computer running. If any operator action is required to physically start the reader, e.g., moving the Teletype reader switch to START, do it now. If you have done everything up to this point correctly the tape will slew (run) through the reader.

This loader will not stop the reader at the end of the tape. The tape will slew all the way through the reader until it runs out. At this point stop the computer and examine the PC. If the PC reads 9E or more the load was not valid. If the load procedure was correct the

PC should contain a value somewhere in your tape read routine. If you get a bad load start the procedure over. The problem of tape error has been eliminated beforehand. Every object tape has been loaded and verified before being sent.

When you have loaded ZEDBUG successfully, follow the instructions for initializing it in appendix C. After ZEDBUG is up and running, you can use the ZEDBUG dump facility to generate a paper tape copy of itself with all of the proper input/output routines. To do this start ZEDBUG, and after the prompt character has appeared (#), type:

D3600 NNNN.

where NNNN is the last address used by your ZEDBUG I/O routines. The computer will pause to allow you to turn on the punch. When you have turned the punch on tap the space bar. Your customized version of ZEDBUG will be punched preceded and followed by a few inches of blank tape. The computer will again pause to allow you to turn the punch off. These pauses are for systems equipped with only a Teletype. Tap the space bar and control will return to the ZEDBUG monitor.

You may now use your customized version of ZEDBUG to initialize the assembler. There is no more need for the special loader in lower memory. Simply put the assembler tape in the reader and type:

L.

in response to the ZEDBUG prompt character (#). Start the reader if necessary (reader switch to START position on Teletype) and the tape will be loaded. If the load is successful it will end with the prompt character. If it is not the word:

CKSM

will be typed, meaning checksum error. Again, tape error has been eliminated by the verification of each tape before it is sent. The trouble may be dirt in the reader or something caught in the holes of the tape. This last can easily occur if the tape is allowed to drag along the floor while being loaded or rewound.

Next go to appendix B, and follow the instructions there for initializing the assembler for your I/O configuration. When you have done this use the ZEDBUG dump facility again to get a tape copy of the assembler customized to your I/O configuration. This is done by typing:

D100 XXXX.

where XXXX is the highest memory location used by your I/O routines. The computer will pause before and after punching as before, waiting for the space bar to be tapped.

If your system is equipped with disk, both ZEDBUG and the assembler can now be copied to disk. Whether or not you have a disk, you should punch the customized paper tape copies. If a disk crash occurs it will save all of the labor of reinitializing. It goes without saying, of course, that a backup copy should be kept of any disk which contains important software or data.



PAGE 01 BOOTSTRAP LOADER

```

002          * THIS ROUTINE INITIALIZES AND STARTS
003          * THE PAPER TAPE READER IN THE WRITER'S
004          * SYSTEM, A REMEX READER CONTROLLED BY
005          * A MOTOROLA 6820.
006          ORG      0
007          INITIALIZE READER
008 0000 AF      SRDR  ZAR      ENTER HERE ON TRP 0
009 0001 D310    OUT   CTLA
010 0003 D312    OUT   CTLB
011 0005 D313    OUT   DTAB
012 0007 3D     DCR   A
013 0008 D311    OUT   DTAA
014 000A 3E04    MVI   A,4
015 000C D310    OUT   CTLA
016 000E 3E06    MVI   A,6
017 0010 D312    OUT   CTLB
018          NOW START THE READER
019 0012 3E0F    MVI   A,:F
020 0014 D311    OUT   DTAA
021 0016 C9     RET    MUST END WITH RETURN
022  CTLA      EQU   :10
023  DTAA      EQU   CTLA+1
024  CTLB      EQU   DTAA+1
025  DTAB      EQU   CTLB+1
026
027          * THIS ROUTINE READS ONE TAPE FRAME
028          * FROM THE REMEX READER AND RETURNS
029          * IT IN A. NOTE THAT ONLY THE A REG.
030          * IS USED, EXCEPT AS SHOWN TO COMPUTE
031          * THE CHECKSUM.
032
033          ORG      :38
034 0038 DB12    PCHR  IN      CTLB      ENTER HERE ON TRP 7
035 003A 17      RAL
036 003B 30FB    $     BNC  PCHR
037 003D DB13    IN      DTAB
038
039          THIS ROUTINE MUST END
040          WITH THESE 5 INSTRUCTIONS
041          WHICH COMPUTE THE
042          TAPE CHECKSUM
043
044 003F F5      PUSH  PSW      SAVE FRAME
045 0040 82      ADD   D        FORM NEW CHECKSUM
046 0041 57      MOV   D,A     SAVE UPDATED CHECKSUM
047 0042 F1      POP   PSW     RETRIEVE FRAME
048 0043 C9      RET    MUST END WITH RETURN
049
050          * WITH THE EXCEPTION OF THE CHECKSUM
051          * SEQUENCE ABOVE, THE ABOVE ROUTINES
052          * WILL BE DIFFERENT FOR YOUR SYSTEM.
053          * THE FOLLOWING WILL BE COMMON TO ALL
054          * SYSTEMS. IT USES THE TWO SPECIAL
055          * USER SUPPLIED ROUTINES AT 0 AND :38
056          * THIS IS THE BOOTSTRAP LOADER PROPER.
057
058          ORG      :80
059 0080 318000  SLOAD  LXI   SP,SLOAD  LOAD STACK POINTER

```

PAGE 02      BOOTSTRAP LOADER

|               |          |      |       |                       |
|---------------|----------|------|-------|-----------------------|
| 060 0083 C7   |          | TRP  | 0     | START READER          |
| 061 0084 FF   | L1       | TRP  | 7     | GET TAPE FRAME        |
| 062 0085 FEFF |          | CPI  | :FF   | CHECK FOR RECORD MARK |
| 063 0087 20FB | \$       | BNE  | L1    | LOOP BACK IF NOT      |
| 064 0089 57   | L2       | MOV  | D.A   | START CHECKSUM        |
| 065 008A FF   |          | TRP  | 7     | GET TAPE FRAME        |
| 066 008B 6F   |          | MOV  | L.A   | LOW ORIGIN            |
| 067 008C FF   |          | TRP  | 7     | GET TAPE FRAME        |
| 068 008D 67   |          | MOV  | H.A   | HIGH ORIGIN           |
| 069 008E FF   |          | TRP  | 7     | GET TAPE FRAME        |
| 070 008F B7   |          | ORA  | A     | CHECK RECORD TYPE     |
| 071 0090 200C | \$       | BNZ  | ERROR | BAD IF NOT ZERO       |
| 072 0092 FF   |          | TRP  | 7     | GET TAPE FRAME        |
| 073 0093 47   |          | MOV  | B.A   | SET RECORD LENGTH     |
| 074 0094 FF   | L3       | TRP  | 7     | GET TAPE FRAME        |
| 075 0095 77   |          | MOV  | M.A   | LOAD DATA RECORD      |
| 076 0096 23   |          | INX  | H     | BUMP POINTER          |
| 077 0097 10FB | \$       | DBNZ | L3    | LOOP TILL FINISHED    |
| 078 0099 5A   |          | MOV  | E.D   | SAVE CHECKSUM         |
| 079 009A FF   |          | TRP  | 7     | GET TAPE CHECKSUM     |
| 080 009B AB   |          | XRA  | E     | COMPARE CHECKSUMS     |
| 081 009C 28E6 | \$       | BZ   | L1    | OK IF ZERO            |
| 082 009E 18FE | \$ ERROR | BRA  | ERROR | OTHERWISE ERROR       |
| 083 00A0      |          | END  |       |                       |

\*\*\*\*\*  
 \* S Y M B O L   T A B L E \*  
 \*\*\*\*\*

|       |      |       |      |      |      |      |      |
|-------|------|-------|------|------|------|------|------|
| CTLA  | 0010 | CTLB  | 0012 | DTAA | 0011 | DTAB | 0013 |
| ERROR | 009E | L1    | 0084 | L2   | 0089 | L3   | 0094 |
| PCHR  | 0038 | SLOAD | 0080 | SRDR | 0000 |      |      |

## APPENDIX B / THE ASSEMBLY PROGRAM

The Z80 resident assembler will function with a wide variety of input/output configurations, from a simple Teletype to high speed paper tape equipment, line printers, CRTs, etc. To use any specific configuration, the assembler must be provided with the input/output routines required by the particular devices. There are 15 possible routines to be initialized. These routines begin at the end of the assembler, location :2465. A jump to each routine is placed in the assembler I/O jump table which begins at :15C (see second page of assembler listing, this appendix). The routines are:

- INTRML    Initialize terminal interface. This routine does whatever is necessary to initialize the device you are using for a terminal. All registers may be used.
- RDRINT    Initialize reader interface. This routine performs any initialization required by the paper tape reader interface. All registers may be used.
- PDVINT    Initialize punch interface. This routine initializes the paper tape punch interface where necessary. All registers may be used.
- LDVCIN    Initialize printing device interface. This routine initializes the interface for whatever device is being used for listing. This is the device on which assembly listings will be produced. All registers may be used.

Note before going further that if only a Teletype is used the functions of all of the above routines may be accomplished by the INTRML routine.

- DTCTCC** Detects Control-C typed on terminal. This routine should return with the carry bit on if Control-C was typed, off otherwise. It should *not* wait in a loop for something to be typed, merely detect whether something has already been typed. If nothing has been typed return with carry off. If something other than Control-C was typed return with carry off. Only the A register may be used.
- RDRSTT** Start the paper tape reader. Only the A register may be used.
- RDRSTP** Stop the paper tape reader. Only the A register may be used.
- GTPCHR** Read a character from the paper tape reader. Use only the A register. The character is returned in the A register, forced to upper case ASCII with the high bit forced to one. The high bit can be forced to one by **ORI :80**.
- SB** Wait for space bar to be struck. Do not return until it has been struck and ignore all other characters. Do not echo the character back to the terminal. Use only A.
- CPNT** Send the character in A to the system listing device. Send the character first, then test for device ready. All registers including A must return unchanged.
- PCHCHR** Punch the character in A on the system punch device. All registers must return unchanged.
- RKB** Read a character from the terminal keyboard. Do not return until something has been typed. The character returned must be in upper case ASCII with the high bit forced to one (**ORI :80**). Return the character read in A. No other registers may be used. To force upper case ASCII check whether the characters is in the :E1-:FA range after the high bit has been set to one. If it is in this range execute **ANI :DF (E6 DF)**.

- OUT** Send the character in A to the terminal device. All registers including A must return unchanged.
- REWIND** This is an optional routine provided for those who wish to modify the assembler for nonstandard devices. It is called by the assembler before each pass.
- GDVCIN** Initialize general device. Also provided for those who may wish to modify the assembler for nonstandard devices. It is called before the first assembler pass.

All of these routines are to be written as subroutines, i.e., they must end with an **RET** instruction. Routines which are not required may consist of nothing but the **RET**.

When the above routines have been created they are to be entered into memory beginning at :2465. The jump table at :15C must be filled in with a jump instruction to the routine named in that position. A **JMP** is a C3. If INTRML began at :2465, the first entry in the jump table would be:

| <i>location</i> | <i>contents</i> |
|-----------------|-----------------|
| 15C             | C3              |
| 15D             | 65              |
| 15E             | 24              |

The jump table is then filled in, beginning at :15C, in the following order:

|            |               |
|------------|---------------|
| <b>JMP</b> | <b>INTRML</b> |
| <b>JMP</b> | <b>RDRINT</b> |
| <b>JMP</b> | <b>PDVINT</b> |
| <b>JMP</b> | <b>LDVCIN</b> |
| <b>JMP</b> | <b>DTCTCC</b> |
| <b>JMP</b> | <b>RDRSTT</b> |
| <b>JMP</b> | <b>RDRSTP</b> |
| <b>JMP</b> | <b>GTPCHR</b> |
| <b>JMP</b> | <b>SB</b>     |
| <b>JMP</b> | <b>CPNT</b>   |
| <b>JMP</b> | <b>PCHCHR</b> |



```

JMP    RKB
JMP    OUT
JMP    REWIND
JMP    GDVCIN

```

Any **JMP** which is not required should be filled in with

```

RET
RES    2

```

which is C9, 00 and 00 in hexadecimal. Note that *all three words are required*.

This is really a good deal simpler than it looks. In a system using only a Teletype requiring no initialization, and having no reader control, the jump table looks like this:

|     |            |               |                                      |
|-----|------------|---------------|--------------------------------------|
| 15C | <b>RET</b> |               | terminal initialize                  |
|     | <b>RES</b> | 2             |                                      |
| 15F | <b>RET</b> |               | reader initialize                    |
|     | <b>RES</b> | 2             |                                      |
| 162 | <b>RET</b> |               | punch initialize                     |
|     | <b>RES</b> | 2             |                                      |
| 165 | <b>RET</b> |               | printer initialize                   |
|     | <b>RES</b> | 2             |                                      |
| 168 | <b>JMP</b> | <b>DTCTCC</b> | detect Control-C                     |
| 16B | <b>RET</b> |               | reader start                         |
|     | <b>RES</b> | 2             |                                      |
| 16E | <b>RET</b> |               | reader stop                          |
|     | <b>RES</b> | 2             |                                      |
| 171 | <b>JMP</b> | <b>GTPCHR</b> | get paper tape frame                 |
| 174 | <b>JMP</b> | <b>SB</b>     | wait for space bar                   |
| 177 | <b>JMP</b> | <b>OUT</b>    | print character on<br>listing device |
| 17A | <b>JMP</b> | <b>OUT</b>    | punch character                      |
| 17D | <b>JMP</b> | <b>RKB</b>    | read from keyboard<br>and echo       |
| 180 | <b>JMP</b> | <b>OUT</b>    | print character<br>on terminal       |

|     |     |   |                              |
|-----|-----|---|------------------------------|
| 183 | RET |   | rewind exotic I/O device     |
|     | RES | 2 |                              |
| 186 | RET |   | initialize exotic I/O device |
|     | RES | 2 |                              |

A system having a more elaborate set of peripherals can use them to advantage. The writer's system has a CRT, high speed paper tape reader and punch and a line printer. As a guide to the general scheme, the initialization of this system is in source form before the assembler command summary in this appendix.

The final adjustment which must be made to the resident assembler is to define the memory space which can be used as source buffer. The address of the beginning of the source buffer is loaded into location :247 and :248, the low order word in :247, high word in :248. The address of the last available buffer word is entered into locations :249 and :24A. The beginning address of the buffer will normally be the first location beyond the end of the assembler's input/output drivers. With the I/O drivers shown for the writer's system this location is :2536. The last available word may be the highest physical memory word. In the writer's system this is :9FFF (40K system), but to protect various utility programs (including ZEDBUG) located at :9000 and beyond this address is set to :8FFF. The four words are thus:

| <i>location</i> | <i>content</i> |
|-----------------|----------------|
| 247             | 36             |
| 248             | 25             |
| 249             | FF             |
| 24A             | 8F             |

in the writer's system. If ZEDBUG were to be protected in a 16K system the address in :249 and :24A would be :3600, the 00 in :249 and the :36 in :24A.

When all of these initialization steps have been taken, set the PC of your computer to :100 and start it. It should respond with the assembler prompt character (>). To test the assembler type in example 4-1 as per the instructions in chapter 5, and try a test assembly. If it is

successful clear the assembler source buffer with a /. command and transfer control to ZEDBUG. An initialized paper tape copy of the assembler can now be punched by typing:

D100 AAAA.

where AAAA is the address of the last word of your input/output. Don't forget that the dump function of ZEDBUG pauses before and after punching to allow you to turn the punch on and off. Tapping the space bar resumes activity.

You now have a working assembler. If you have a disk this is the time to make a disk copy.

### Summary of Editor/Assembler Commands

The editor/assembler indicates its readiness to accept a command by typing the prompt character. All commands are terminated with a period. Use of the letter F in place of a line number indicates the last line of the source buffer. Any illegal command will draw "?" as a response, and the command will be ignored.

/. Initialize. This clears the source buffer.

An. Add text lines after line n in the existing text. If no line number is given, or if n is given as zero, the text typed in will be added before any existing text. Each line is terminated by a carriage return. The A command itself is terminated by using a period as the first character of a text line.

Dn. or Dn m. Deletes line n or lines n through m inclusive.

Ln. or Ln m. Lists line n or lines n through m on the system terminal device. Each listed line is preceded by a four digit line number and is formatted into appropriate fields.

- Pn. or Pn m. Causes line n or lines n through m to be punched on the system punch device. The execution of the P command is preceded and followed by a pause to allow the punch device to be turned on and off. This is required when the punch and terminal devices are the same physical peripheral, e.g., a Teletype. The text lines are punched with no superfluous blanks and the text is preceded and followed by several inches of blank tape.
- R. Reads text lines from the system paper tape read device and adds them to the end of what is already in the buffer. No reader start/stop control is necessary to use this command.
- #L. Assembles the program in the source buffer and prints the assembly listing on the system listing device.
- #O. Assembles the program in the source buffer and punches an object tape on the system punch device. The message READY PUNCH, SPACE will appear followed by a pause. Press the space bar to begin punching. When punching has finished another pause is executed. Press space to return to command mode.
- #E. Assembles the program in the source buffer printing only lines which contain errors on the system terminal device. Errors uncovered by this error scan process can then be corrected by the D and A commands of the text editor.
- #Z. Assembles the program in the source buffer printing lines which contain Z80 only instructions and lines containing errors. This mode is provided as a convenient means of flagging Z80 only instructions in programs which must retain 8080 compatibility.

**Assembler Error Flags**

- P Symbol which appears in the operand field of an **EQU**, **RES** or **ORG** has not been defined previous to its appearance as an operand.
- S A symbol is used as a label which violates the rules for symbol formation. A symbol must begin with an alphabetic character, be from one to six characters in length and may not contain +, - or comma.
- A A branch instruction (relative jump) attempts to address a point outside the branch range PC+127 to PC-128.
- M Multiple symbol definition. The same symbol has been used more than once as a label. Note that this will occur if two symbols exceed the six character length but have the same first six characters. The assembler ignores characters after the first six.
- L The length of the operand of this instruction exceeds the legal length for this instruction.
- U The operand of this instruction or pseudo-op is undefined or contains an undefined element.
- R Register specification error.
- O The instruction field does not contain a legal 8080 or Z80 mnemonic or a legal pseudo-op.
- D The displacement field of an indexed instruction contains an undefined element or generates data which cannot be held in eight bits.

**Assembling Programs Too Long To Be Held In Memory**

Programs too long to be held in available memory may be assembled directly from paper tape. Reader on/off control is necessary to



use this option. Assembly from tape is accomplished by typing T as the second character of the assemble command, i.e.:

- #TL. Assemble for listing.
- #TO. Assemble for object tape.
- #TE. Assemble and print errors only.
- #TZ. Scan for Z80 only instructions.

Mount the tape on the system paper tape reader and type the selected command. The tape will slew through the reader and stop. Remount the tape and press the space bar. The tape will again slew through the reader and the program will assemble as per the assembly specification, L, O, E or Z.

Tape programs may be segmented by making the last line of each tape segment except the final one begin with an exclamation point. This line will not enter into the assembly but will stop the reader and cause a pause to load the next tape segment. Pressing space will resume tape reading. The last tape segment terminates with the **END** statement and no exclamation point.

Note that in assembling very long programs (like the assembler itself) there will be a delay of some seconds between the two passes of the tape. This is because the symbol table is being sorted. Mount the tape for the second pass and tap the space bar. When the symbol table sort is finished the reader will start.

PAGE 01 I/O ASSIGNMENTS FOR Z80 ASSEMBLER

```

002                                ORG    :15C
003 015C C36524                    JMP    INTRML
004 015F C36E24                    JMP    RDRINT
005 0162 C38424                    JMP    PDVINT
006 0165 C3D724                    JMP    LDYCIN
007 0168 C39524                    JMP    DTCTCC
008 016B C3A724                    JMP    RDRSTT
009 016E C3AC24                    JMP    RDRSTP
010 0171 C3B124                    JMP    GTPCHR
011 0174 C3BC24                    JMP    SB
012 0177 C3CC24                    JMP    CPNT
013 017A C30025                    JMP    PCHCHR
014 017D C31C25                    JMP    RKB
015 0180 C32A25                    JMP    OUT
016 0183 C9                        RET
017 0184                            RES    2
018 0186 C9                        RET
019 0187                            RES    2
020
021
022                                ORG    :2465
023                                * TERMINAL DEVICE INITIALIZATION
024 2465 3E03                      INTRML  MVI    A,3
025 2467 D320                      OUT     STAT
026 2469 3E11                      MVI    A,:11
027 246B D320                      OUT     STAT
028 246D C9                        RET
029                                STAT  EQU    :20
030                                DATA  EQU    STAT+1
031
032                                * TAPE READER INTERFACE INITIALIZE
033 246E AF                        RDRINT  ZAR
034 246F D310                      OUT     CTLA
035 2471 D312                      OUT     CTLB
036 2473 D313                      OUT     DTAB
037 2475 3D                        DCR     A
038 2476 D311                      OUT     DTAA
039 2478 3E04                      MVI    A,4
040 247A D310                      OUT     CTLA
041 247C 3E06                      MVI    A,6
042 247E D312                      OUT     CTLB
043 2480 CDAC24                    CALL   RDRSTP
044 2483 C9                        RET
045                                CTLA  EQU    :10
046                                DTAA  EQU    CTLA+1
047                                CTLB  EQU    DTAA+1
048                                DTAB  EQU    CTLB+1
049
050                                * PUNCH DEVICE INITIALIZATION
051 2484 AF                        PDVINT  ZAR
052 2485 D314                      OUT     CTA
053 2487 D316                      OUT     CTB
054 2489 3D                        DCR     A
055 248A D317                      OUT     DTB
056 248C 3E04                      MVI    A,4
057 248E D314                      OUT     CTA
058 2490 3E36                      MVI    A,:36
059 2492 D316                      OUT     CTB

```

PAGE 02 I/O ASSIGNMENTS FOR Z80 ASSEMBLER

```

060 2494 C9          RET
061
062          * DETECT CONTROL-C
063
064          RETURNS WITH CARRY ON IF
065          CONTROL-C WAS TYPED, OFF
          OTHERWISE.
066 2495 DB20      DTCTCC  IN      STAT
067 2497 1F          RAR
068 2498 D0          RNC
069 2499 DB21          IN      DATA
070 249B F680          ORI      :80
071 249D FE83          CPI      :83
072 249F C2A424       JNE      DTC2
073 24A2 37          STC
074 24A3 C9          RET
075 24A4 B7          DTC2  RSC
076 24A5 C9          RET
077 24A6 C9          RET
078
079          * START PAPER TAPE READER
080 24A7 3E0F      RDRSTT  MVI      A,GOBITS
081 24A9 D311          OUT      DTAA
082 24AB C9          RET
083
084          * STOP PAPER TAPE READER
085 24AC 3EF0      RDRSTP  MVI      A,STPBIT
086 24AE D311          OUT      DTAA
087 24B0 C9          RET
088          GOBITS  EQU      :F
089          STPBIT  EQU      :F0
090
091          * GET CHARACTER FROM PAPER TAPE READER
092 24B1 DB12      GTPCHR  IN      CTLB
093 24B3 17          RAL
094 24B4 D2B124     JNC      GTPCHR
095 24B7 DB13          IN      DTAB
096 24B9 F680          ORI      :80
097 24BB C9          RET
098
099          * WAIT FOR SPACE BAR
100 24BC DB20      SB      IN      STAT
101 24BE 1F          RAR
102 24BF D2BC24     JNC      SB
103 24C2 DB21          IN      DATA
104 24C4 F680          ORI      :80
105 24C6 FEA0          CPI      ' '
106 24C8 C2BC24     JNE      SB
107 24CB C9          RET
108
109          * PRINT CHARACTER ON LISTING DEVICE
110 24CC D31F      CPNT    OUT      DB
111 24CE DB1E      CP2     IN      CB
112 24D0 17          RAL
113 24D1 D2CE24     JNC      CP2
114 24D4 DB1F          IN      DB
115 24D6 C9          RET
116
117          * INITIALIZE PRINTER INTERFACE

```

PAGE 03 I/O ASSIGNMENTS FOR Z80 ASSEMBLER

```

118 24D7 AF          LDYCIN ZAR
119 24D8 D31C        OUT    CA
120 24DA D31E        OUT    CB
121 24DC 3D          DCR    A
122 24DD D31F        OUT    DB
123 24DF 3E01        MVI    A,1
124 24E1 D31D        OUT    DA
125 24E3 3E04        MVI    A,4
126 24E5 D31C        OUT    CA
127 24E7 AF          ZAR
128 24E8 D31D        OUT    DA
129 24EA 3E01        MVI    A,1
130 24EC D31D        OUT    DA
131 24EE 3E2C        MVI    A,:2C
132 24F0 D31E        OUT    CB
133 24F2 DB1F        IN     DB
134 24F4 3EA0        MVI    A,' '
135 24F6 0650        MVI    B,80
136 24F8 CDCC24      LDYC2  CALL CPNT
137 24FB 05          DCR    B
138 24FC C2F824      JNZ   LDYC2
139 24FF C9          RET
140                  CA     EQU   :1C
141                  DA     EQU   CA+1
142                  CB     EQU   DA+1
143                  DB     EQU   CB+1
144
145                  * PUNCH CHARACTER ON PUNCH DEVICE
146 2500 D317        PCHCHR OUT   DTB
147 2502 DB15        PCH2  IN    DTA
148 2504 E681        ANI    :81
149 2506 C20225      JNZ   PCH2
150 2509 3E3E        MVI    A,:3E
151 250B D316        OUT    CTB
152 250D 3E36        MVI    A,:36
153 250F D316        OUT    CTB
154 2511 DB17        IN     DTB
155 2513 DB16        PCH3  IN    CTB
156 2515 17          RAL
157 2516 D21325      JNC   PCH3
158 2519 DB17        IN     DTB
159 251B C9          RET
160                  CTA     EQU   :14
161                  DTA     EQU   CTA+1
162                  CTB     EQU   DTA+1
163                  DTB     EQU   CTB+1
164
165                  * READ CHARACTER FROM TERMINAL DEVICE
166 251C DB20        RKB    IN    STAT
167 251E 1F          RAR
168 251F D21C25      JNC   RKB
169 2522 DB21        IN     DATA
170 2524 F680        ORI    :80
171 2526 CD2A25      CALL  OUT
172 2529 C9          RET
173
174                  * OUTPUT CHARACTER TO TERMINAL DEVICE
175 252A D321        OUT    OUT    DATA

```

PAGE 04 I/O ASSIGNMENTS FOR Z80 ASSEMBLER

|                 |       |      |       |                   |
|-----------------|-------|------|-------|-------------------|
| 176 252C F5     |       | PUSH | PSW   |                   |
| 177 252D DB20   | OUT2  | IN   | STAT  |                   |
| 178 252F IF     |       | RAR  |       |                   |
| 179 2530 IF     |       | RAR  |       |                   |
| 180 2531 D22D25 |       | JNC  | OUT2  |                   |
| 181 2534 F1     |       | POP  | PSW   |                   |
| 182 2535 C9     |       | RET  |       |                   |
| 183 2536 00     | E01/0 | ***  |       |                   |
| 184             |       | ORG  | :247  |                   |
| 185 0247 3625   |       | DBL  | E01/0 | INITIALIZE BUFFER |
| 186 0249        |       | END  |       |                   |

\*\*\*\*\*  
 \* S Y M B O L   T A B L E \*  
 \*\*\*\*\*

|        |      |        |      |        |      |        |      |
|--------|------|--------|------|--------|------|--------|------|
| CA     | 001C | CB     | 001E | CP2    | 24CE | CPNT   | 24CC |
| CTA    | 0014 | CTB    | 0016 | CTLA   | 0010 | CTLB   | 0012 |
| DA     | 001D | DATA   | 0021 | DB     | 001F | DTA    | 0015 |
| DTAA   | 0011 | DTAB   | 0013 | DTB    | 0017 | DTC2   | 24A4 |
| DTCTCC | 2495 | E01/0  | 2536 | GOBITS | 000F | GTPCHR | 24B1 |
| INTRML | 2465 | LDVC2  | 24F8 | LDVCIN | 24D7 | OUT    | 252A |
| OUT2   | 252D | PCH2   | 2502 | PCH3   | 2513 | PCHCHR | 2500 |
| PDVINT | 2484 | RDRINT | 246E | RDRSTP | 24AC | RDRSTT | 24A7 |
| RKB    | 251C | SB     | 24BC | STAT   | 0020 | STPBIT | 00F0 |



## ASSEMBLY LISTING OF THE ASSEMBLY PROGRAM

```

002 *****
003 *
004 * RESIDENT ASSEMBLER FOR Z80 BASED SYSTEMS *
005 *
006 * COPYRIGHT WALTER J. WELLER MAY 1978 *
007 *
008 *****
009 *
010 * "I BEHELD THE WRETCH - THE MISERABLE MONSTER
011 * WHOM I HAD CREATED."
012 * FRANKENSTEIN
013 * MARY WOLLSTONECRAFT SHELLEY
014 *
015 *
016
017 0100 310001 MON ORG :100 SP,MON ENTRY
018 0103 CD5C01 CALL INTRML INITIALIZE TERMINAL INTERFAC
019 0106 CD5F01 CALL RDRINT INITIALIZE PTR INTERFACE
020 0109 CD6201 CALL PDVINT INITIALIZE PUNCH INTERFACE
021 010C CD6501 CALL LDVCIN INITIALIZE LISTING DEVICE
022 010F 3A5902 LDA FLAG
023 0112 B7 ORA A INITIALIZED YET?
024 0113 C2F302 JNZ CLR2 NOT IF NON ZERO
025 0116 CD8601 MON1 CALL GDVCIN
026 0119 310001 MON2 LXI SP,MON
027 011C CD8901 CALL CRLF
028 011F 3EBE MVI A,'>'
029 0121 CD8001 CALL OUT PRINT PROMPT CHARACTER
030 0124 CD7D01 CALL RKB GET COMMAND
031 0127 FEF7 CPI '/'
032 0129 CAEB02 JEQ CLEAR
033 012C FECC CPI 'L'
034 012E CA1103 JEQ LIST
035 0131 FEC4 CPI 'D'
036 0133 CAF203 JEQ DELETE
037 0136 FEC1 CPI 'A'
038 0138 CA3B04 JEQ ADD
039 013B FED2 CPI 'R'
040 013D CA3E05 JEQ READ
041 0140 FED0 CPI 'P'
042 0142 CA0903 JEQ PUNCH
043 0145 FEA3 CPI '#'
044 0147 CA5E10 JEQ ASM
045
046 EXPANSION COMMANDS
047
048 014A FE00 CPI 0
049 014C CA0000 JEQ 0
050 014F FE00 CPI 0
051 0151 CA0000 JEQ 0
052 0154 FE00 CPI 0
053 0156 CA0000 JEQ 0
054 0159 C3E002 JMP ERR
055 *
056 *
057 *****
058 * INPUT/OUTPUT CONTROL ROUTINES *
059 *****

```

```

060      *
061      * INITIALIZE TERMINAL DEVICE INTERFACE
062 015C C9  INTRML RET
063 015D      RES 2
064      *
065      * INITIALIZE PAPER TAPE READER INTERFACE
066 015F C9  RDRINT RET
067 0160      RES 2
068      *
069      * INITIALIZE PUNCH DEVICE INTERFACE
070 0162 C9  PDVINT RET
071 0163      RES 2
072      *
073      * INITIALIZE LISTING DEVICE INTERFACE
074 0165 C9  LDVCIN RET
075 0166      RES 2
076      *
077      * DETECT CONTROL-C TYPED ON TERMINAL DEVICE
078 0168 C9  DTCTCC RET
079 0169      RES 2
080      *
081      * START PAPER TAPE READER
082 016B C9  RDRSTT RET
083 016C      RES 2
084      *
085      * STOP PAPER TAPE READER
086 016E C9  RDRSTP RET
087 016F      RES 2
088      *
089      * GET CHARACTER FROM PAPER TAPE READER
090 0171 C9  GTPCHR RET
091 0172      RES 2
092      *
093      * WAIT FOR SPACE ON TERMINAL DEVICE
094 0174 C9  SB RET
095 0175      RES 2
096      *
097      * SEND ONE CHARACTER TO LISTING DEVICE
098 0177 C9  CPNT RET
099 0178      RES 2
100      *
101      * PUNCH CHARACTER ON PUNCH DEVICE
102 017A C9  PCHCHR RET
103 017B      RES 2
104      *
105      * READ CHARACTER FROM KEYBOARD AND ECHO
106 017D C9  RKB RET
107 017E      RES 2
108      *
109      * OUTPUT ONE CHARACTER TO TERMINAL DEVICE
110 0180 C9  OUT RET
111 0181      RES 2
112      *
113      * REWIND/RESET GENERAL INPUT/OUTPUT DEVICE
114 0183 C9  REWIND RET
115 0184      RES 2
116      *
117      * INITIALIZE GENERAL DEVICE

```

PAGE 03 Z80 RESIDENT EDITOR/ASSEMBLER

```

118 0186 C9      GDVCIN  RET
119 0187          RES    2
120              *****
121              *
122              * OUTPUT CARRIAGE RETURN AND LINE FEED
123              *
124 0189 3E8D      CRLF    MVI    A,:8D
125 018B CD8001    CALL    OUT
126 018E 3E8A      MVI    A,:8A
127 0190 CD8001    CALL    OUT
128 0193 3EFF      MVI    A,:FF
129 0195 CD8001    CALL    OUT
130 0198 C9        RET
131              *
132              * PRINT THE LINE WHOSE BEGINNING ADDRESS
133              * IS IN H&L ON ENTRY. C.R./LF. PRECEED
134              * EVERY LINE. LINE TERMINATES WHEN
135              * CARRIAGE RETURN IS ENCOUNTERED.
136              *
137 0199 CD8901    PRT      CALL    CRLF      RETURN CARRIAGE
138 019C 7E        PRT2     MOV     A,M        GET CHARACTER
139 019D FE8D      CPI      :8D
140 019F C8        RZ
141 01A0 CD8001    CALL    OUT
142 01A3 23        INC     H
143 01A4 C39C01    JMP     PRT2
144              *
145              * FETCH DECIMAL NUMBER FROM TTY
146              *
147 01A7 210000    IDC      LXI     H,0
148 01AA CD7D01    IDC2     CALL    RKB
149 01AD FEB0      CPI      :B0
150 01AF DACF01    JAL      IDCRTN
151 01B2 FEB8      CPI      :BA
152 01B4 D2CF01    JGE      IDCRTN
153 01B7 E60F      ANI      :F
154 01B9 4F        MOV     C,A
155 01BA AF        ZAR
156 01BB 47        MOV     B,A
157 01BC 29        DAD     H
158 01BD 17        RAL
159 01BE 54        MOV     D,H
160 01BF 5D        MOV     E,L
161 01C0 29        DAD     H
162 01C1 17        RAL
163 01C2 29        DAD     H
164 01C3 17        RAL
165 01C4 19        DAD     D
166 01C5 17        RAL
167 01C6 09        DAD     B
168 01C7 17        RAL
169 01C8 B7        ORA     A
170 01C9 CAAA01    JZ      IDC2
171 01CC C3E002    JMP     ERR
172 01CF C9        IDCRTN  RET
173              *
174              * FETCH ONE DECIMAL NUMBER FROM KEYBOARD
175              *

```

|                 |       |      |        |                            |
|-----------------|-------|------|--------|----------------------------|
| 176 01D0 CDA701 | PFCH  | CALL | IDC    |                            |
| 177 01D3 FEC6   |       | CPI  | 'F'    | LAST LINE?                 |
| 178 01D5 C0     |       | RNZ  |        |                            |
| 179 01D6 2A3B02 |       | LHLD | LNCNT  | GET # OF LAST LINE         |
| 180 01D9 CD7D01 |       | CALL | RKB    | AND NEXT CHARACTER         |
| 181 01DC C9     |       | RET  |        |                            |
| 182             | *     |      |        |                            |
| 183             | *     |      |        |                            |
| 184             | *     |      |        |                            |
| 185             | *     |      |        |                            |
| 186             | *     |      |        |                            |
| 187             | *     |      |        |                            |
| 188             | *     |      |        |                            |
| 189             | *     |      |        |                            |
| 190             | *     |      |        |                            |
| 191 01DD CDD001 | FPRM  | CALL | PFCH   | GET DECIMAL NUMBER         |
| 192 01E0 F5     |       | PUSH | PSW    | SAVE LAST CHARACTER        |
| 193 01E1 7C     |       | MOV  | A,H    | TEST FOR ZERO              |
| 194 01E2 B5     |       | ORA  | L      |                            |
| 195 01E3 C2EA01 |       | JNZ  | FPI    |                            |
| 196 01E6 F1     |       | POP  | PSW    |                            |
| 197 01E7 C3E002 |       | JMP  | ERR    |                            |
| 198 01EA F1     | FPI   | POP  | PSW    |                            |
| 199 01EB 224F02 |       | SHLD | FIRST  | SAVE FIRST LINE #          |
| 200 01EE FEAE   |       | CPI  | ' '    | PERIOD?                    |
| 201 01F0 CA0002 |       | JEQ  | FPRM2  | YES, SAVE AS SECOND LINE # |
| 202 01F3 FEA0   | FPRM1 | CPI  | ' '    | BLANK?                     |
| 203 01F5 C2E002 |       | JNE  | ERR    |                            |
| 204 01F8 CDD001 |       | CALL | PFCH   | GET SECOND LINE #          |
| 205 01FB FEAE   |       | CPI  | ' '    | PERIOD                     |
| 206 01FD C2E002 |       | JNE  | ERR    | ERROR IF NOT               |
| 207 0200 225102 | FPRM2 | SHLD | SECOND | SAVE SECOND LINE #         |
| 208 0203 EB     |       | XCHG |        |                            |
| 209 0204 7A     |       | MOV  | A,D    |                            |
| 210 0205 2F     |       | CMA  |        |                            |
| 211 0206 57     |       | MOV  | D,A    | FORM ONE'S COMPLEMENT      |
| 212 0207 7B     |       | MOV  | A,E    |                            |
| 213 0208 2F     |       | CMA  |        |                            |
| 214 0209 5F     |       | MOV  | E,A    |                            |
| 215 020A 2A4F02 |       | LHLD | FIRST  |                            |
| 216 020D 19     |       | DAD  | D      |                            |
| 217 020E AF     |       | ZAR  |        |                            |
| 218 020F 84     |       | ADD  | H      |                            |
| 219 0210 F2E002 |       | JP   | ERR    | FIRST > SECOND, ERROR      |
| 220 0213 225302 |       | SHLD | IDIFF  | SAVE INCLUSIVE DIFFERENCE  |
| 221 0216 2A3B02 |       | LHLD | LNCNT  |                            |
| 222 0219 19     |       | DAD  | D      |                            |
| 223 021A 23     |       | INX  | H      |                            |
| 224 021B AF     |       | ZAR  |        |                            |
| 225 021C 84     |       | ADD  | H      |                            |
| 226 021D FAE002 |       | JM   | ERR    |                            |
| 227 0220 2A4F02 |       | LHLD | FIRST  |                            |
| 228 0223 CDAB02 |       | CALL | FBA    | GET ADDRESSES FOR 1ST LINE |
| 229 0226 223D02 |       | SHLD | BADR1  | SAVE BEGINNING ADDRESS     |
| 230 0229 EB     |       | XCHG |        |                            |
| 231 022A 223F02 |       | SHLD | EADR1  | SAVE ENDING ADDRESS        |
| 232 022D 2A5102 |       | LHLD | SECOND |                            |
| 233 0230 CDAB02 |       | CALL | FBA    |                            |



PAGE 05 Z80 RESIDENT EDITOR/ASSEMBLER

```

234 0233 224102          SHLD  BADR2      SAVE BEGINNING ADDRESS
235 0236 EB             XCHG
236 0237 224302          SHLD  EADR2      SAVE ENDING ADDRESS
237 023A C9             RET
238
239                      *
240                      *****
241                      * DATA AREA
242                      *****
243                      *
243 023B 0000          LNCNT  DBL  0
244 023D 0000          BADR1  DBL  0
245 023F 0000          EADR1  DBL  0
246 0241 0000          BADR2  DBL  0
247 0243 0000          EADR2  DBL  0
248 0245 0000          BFA    DBL  0
249 0247 0000          BFR    DBL  0
250 0249 FF8F          EOB     DBL  :8FFF      INITIALIZED BY USER!!!
251 024B 0000          LADR    DBL  0
252 024D 0000          EOLADR  DBL  0
253 024F 0000          FIRST  DBL  0
254 0251 0000          SECOND  DBL  0
255 0253 0000          IDIFF  DBL  0
256 0255 0000          NXTCHR DBL  0
257 0257 0000          TEMP   DBL  0
258 0259 01           FLAG    DATA 1
259 025A 00           LASTIN  ***
260 025B 3C           LDRSIZ  DATA 60
261 025C 50           TRLSIZ  DATA 80
262 025D 00           PFLAG   ***
263 025E 00           ACNT     ***
264 025F 00           NULLS    ***
265 0260 0000          DCNT    DBL  0
266 0262             KB       RES  73
267
268                      *
269                      *****
270                      *
271                      * FIND BUFFER ADDRESSES FOR LINE WHOSE NUMBER
272                      * IS IN H&L ON ENTRY. ADDRESS OF 1ST CHARACTER
273                      * IS LEFT IN LADR, ADDRESS OF TERMINATING C.R.
274                      * IS LEFT IN EOLADR, BEGINNING ADDRESS
275                      * IS ALSO IN H&L, ENDING ADDRESS IN D&E.
276                      *
276 02AB EB          FBA      XCHG
277 02AC 2A4502      LHL      BFA      ADDRESS OF START OF BUFFER
278 02AF 7A          MOV      A,D      CHECK FOR ZERO LINE #
279 02B0 B3          ORA      E
280 02B1 C2BB02      JNZ      FBA1
281 02B4 224B02      SHLD     LADR
282 02B7 2B          DCX      H
283 02B8 C3D802      JMP      FBA5
284 02BB 1B          FBA1     DCX      D
285 02BC 7A          MOV      A,D
286 02BD B3          ORA      E      CHECK FOR ZERO COUNT
287 02BE CACB02      JZ       FBA3
288 02C1 7E          FBA2     MOV      A,M
289 02C2 23          INX      H
290 02C3 FE8D          CPI      :8D
291 02C5 CAB002      JEQ      FBA1

```

```

292 02C8 C3C102      JMP FBA2
293 02CB 224B02      FBA3 SHLD LADR
294 02CE 7E          FBA4 MOV A,M
295 02CF FE8D        CPI :8D
296 02D1 CAD802      JEQ FBA5
297 02D4 23          INX H
298 02D5 C3CE02      JMP FBA4
299 02D8 224D02      FBA5 SHLD EOLADR
300 02DB EB          XCHG
301 02DC 2A4B02      LHL LADR
302 02DF C9          RET
303
304 * ERROR ENTRY
305 *
306 02E0 3EBF        ERR MVI A,'?'
307 02E2 CD8001      CALL OUT
308 02E5 310001      LXI SP,MON
309 02E8 C31901      JMP MON2
310
311 * INITIALIZATION
312 *
313 02EB CD7D01      CLEAR CALL RKB      WAIT FOR PERIOD
314 02EE FEAE        CPI '.'
315 02F0 C2E002      JNE ERR
316 02F3 2A4702      CLR2 LHL BFR      GET BUFFER ADDRESS
317 02F6 224502      SHLD BFA
318 02F9 225502      SHLD NXTCHR    SET POINTER TO B.O.B.
319 02FC 210000      LXI H,0
320 02FF 223B02      SHLD LNCNT    ZAP LINE COUNT
321 0302 AF          ZAR
322 0303 325902      STA FLAG      CLEAR START FLAG
323 0306 C31901      JMP MON2
324
325 * LIST AND PUNCH FUNCTIONS
326 *
327 0309 CD8601      PUNCH CALL GDVCIN
328 030C 3E01        MVI A,1
329 030E C31203      JMP L2
330 0311 AF          LIST ZAR
331 0312 325D02      L2 STA PFLAG    SET LIST/PUNCH FLAG
332 0315 CD0D01      CALL FPRM    GET LIMITS
333 0318 3A5D02      LDA PFLAG    LIST OR PUNCH
334 031B B7          ORA A
335 031C CA3403      JZ L3A
336 031F 215E11      LXI H,RDPMSG
337 0322 CD9901      CALL PRT
338 0325 3A5B02      LDA LDRSIZ
339 0328 47          MOV B,A      GET LEADER LENGTH
340 0329 CD7401      CALL SB      WAIT FOR SPACE BAR
341 032C AF          ZAR
342 032D CD7A01      L3 CALL PCHCHR
343 0330 05          DCR B      PUNCH LEADER
344 0331 C22D03      JNZ L3
345 0334 CD8901      L3A CALL CRLF
346 0337 CD6801      L4 CALL DTCTCC
347 033A DA1901      JC MON2
348 033D 2A4F02      LHL FIRST
349 0340 23          INX H      GET LINE NUMBER
                                INCREMENT...

```

|                 |      |      |         |                      |
|-----------------|------|------|---------|----------------------|
| 350 0341 224F02 |      | SHLD | FIRST   | AND REPLACE          |
| 351 0344 2B     |      | DCX  | H       |                      |
| 352 0345 3A5D02 |      | LDA  | PFLAG   |                      |
| 353 0348 B7     |      | ORA  | A       | CONVERT LINE NUMBER? |
| 354 0349 C27A03 |      | JNZ  | L5      |                      |
| 355 034C 1118FC |      | LXI  | D,-1000 |                      |
| 356 034F CDD703 |      | CALL | CONVI   |                      |
| 357 0352 119CFF |      | LXI  | D,-100  |                      |
| 358 0355 CDD703 |      | CALL | CONVI   |                      |
| 359 0358 11F6FF |      | LXI  | D,-10   |                      |
| 360 035B CDD703 |      | CALL | CONVI   |                      |
| 361 035E 11FFFF |      | LXI  | D,-1    |                      |
| 362 0361 CDD703 |      | CALL | CONVI   |                      |
| 363 0364 3EA0   |      | MVI  | A,' '   | PRINT BLANK          |
| 364 0366 CD8001 |      | CALL | OUT     |                      |
| 365 0369 2A3D02 |      | LHLD | BADR1   |                      |
| 366 036C CD2106 |      | CALL | FORMAT  |                      |
| 367 036F 219D07 |      | LXI  | H,LABEL |                      |
| 368 0372 3E8D   |      | MVI  | A,:8D   |                      |
| 369 0374 32D107 |      | STA  | HEX+72  |                      |
| 370 0377 C37D03 |      | JMP  | L5A     |                      |
| 371 037A 2A3D02 | L5   | LHLD | BADR1   |                      |
| 372 037D 3A5D02 | L5A  | LDA  | PFLAG   |                      |
| 373 0380 B7     |      | ORA  | A       |                      |
| 374 0381 CA9603 |      | JZ   | L5AB    |                      |
| 375 0384 7E     | L5A1 | MOV  | A,M     |                      |
| 376 0385 CD7A01 |      | CALL | PCHCHR  |                      |
| 377 0388 23     |      | INX  | H       |                      |
| 378 0389 FE8D   |      | CPI  | :8D     |                      |
| 379 038B C28403 |      | JNE  | L5A1    |                      |
| 380 038E 3E8A   |      | MVI  | A,:8A   |                      |
| 381 0390 CD7A01 |      | CALL | PCHCHR  |                      |
| 382 0393 C3AF03 |      | JMP  | L5C     |                      |
| 383 0396 7E     | L5AB | MOV  | A,M     |                      |
| 384 0397 CD8001 |      | CALL | OUT     |                      |
| 385 039A 23     |      | INX  | H       |                      |
| 386 039B FE8D   |      | CPI  | :8D     | CARRIAGE RETURN      |
| 387 039D C29603 |      | JNE  | L5AB    |                      |
| 388 03A0 3E8A   |      | MVI  | A,:8A   |                      |
| 389 03A2 CD8001 |      | CALL | OUT     |                      |
| 390 03A5 2A3D02 |      | LHLD | BADR1   |                      |
| 391 03A8 7E     | L5B  | MOV  | A,M     |                      |
| 392 03A9 23     |      | INX  | H       |                      |
| 393 03AA FE8D   |      | CPI  | :8D     |                      |
| 394 03AC C2A803 |      | JNE  | L5B     |                      |
| 395 03AF 223D02 | L5C  | SHLD | BADR1   |                      |
| 396 03B2 2A5302 |      | LHLD | IDIFF   |                      |
| 397 03B5 23     |      | INX  | H       |                      |
| 398 03B6 225302 |      | SHLD | IDIFF   |                      |
| 399 03B9 7C     |      | MOV  | A,H     |                      |
| 400 03BA B7     |      | ORA  | A       |                      |
| 401 03BB FA3703 |      | JM   | L4      |                      |
| 402 03BE 3A5D02 |      | LDA  | PFLAG   |                      |
| 403 03C1 B7     |      | ORA  | A       |                      |
| 404 03C2 CAD403 |      | JZ   | L7      |                      |
| 405 03C5 3A5C02 |      | LDA  | TRLSIZ  | GET TRAILER SIZE     |
| 406 03C8 47     |      | MOV  | B,A     |                      |
| 407 03C9 AF     |      | ZAR  |         |                      |

|                 |                   |      |        |                            |
|-----------------|-------------------|------|--------|----------------------------|
| 408 03CA CD7A01 | L6                | CALL | PCHCHR |                            |
| 409 03CD 05     |                   | DCR  | B      |                            |
| 410 03CE C2CA03 |                   | JNZ  | L6     |                            |
| 411 03D1 CD7401 |                   | CALL | SB     |                            |
| 412 03D4 C31601 | L7                | JMP  | MON1   |                            |
| 413             | *                 |      |        |                            |
| 414             | * CONVERTER       |      |        |                            |
| 415             | *                 |      |        |                            |
| 416 03D7 CDDE03 | CONV1             | CALL | CNVDEC |                            |
| 417 03DA CD8001 |                   | CALL | OUT    |                            |
| 418 03DD C9     |                   | RET  |        |                            |
| 419 03DE 06B0   | CNVDEC            | MVI  | B,'0'  |                            |
| 420 03E0 225702 | CONV1A            | SHLD | TEMP   |                            |
| 421 03E3 19     |                   | DAD  | D      |                            |
| 422 03E4 7C     |                   | MOV  | A,H    |                            |
| 423 03E5 B7     |                   | ORA  | A      |                            |
| 424 03E6 FAED03 |                   | JM   | CONV2  |                            |
| 425 03E9 04     |                   | INR  | B      |                            |
| 426 03EA C3E003 |                   | JMP  | CONV1A |                            |
| 427 03ED 2A5702 | CONV2             | LHLD | TEMP   |                            |
| 428 03F0 78     |                   | MOV  | A,B    |                            |
| 429 03F1 C9     |                   | RET  |        |                            |
| 430             | *                 |      |        |                            |
| 431             | * DELETE FUNCTION |      |        |                            |
| 432             | *                 |      |        |                            |
| 433 03F2 CDD001 | DELETE            | CALL | FRM    | GET DELETE LIMITS          |
| 434 03F5 2A3B02 |                   | LHLD | LNCNT  |                            |
| 435 03F8 EB     |                   | XCHG |        |                            |
| 436 03F9 2A5302 |                   | LHLD | IDIFF  |                            |
| 437 03FC 19     |                   | DAD  | D      | COMPUTE NEW LINE COUNT     |
| 438 03FD 223B02 |                   | SHLD | LNCNT  |                            |
| 439 0400 2A5102 |                   | LHLD | SECOND |                            |
| 440 0403 CD3304 |                   | CALL | NEGHL  |                            |
| 441 0406 19     |                   | DAD  | D      | COMPUTE LINECOUNT - SECOND |
| 442 0407 7C     |                   | MOV  | A,H    |                            |
| 443 0408 B5     |                   | ORA  | L      | TEST FOR ZERO              |
| 444 0409 C21504 |                   | JNZ  | DL1    | INCLUDE LAST LINE?         |
| 445 040C 2A3D02 |                   | LHLD | BADR1  | YES, SET NEW END           |
| 446 040F 225502 | DL0               | SHLD | NXTCHR |                            |
| 447 0412 C31901 |                   | JMP  | MON2   |                            |
| 448 0415 2A4302 | DL1               | LHLD | EADR2  | ADDRESS OF FIRST CHAR      |
| 449 0418 23     |                   | INX  | H      | OF LINE TO BE MOVED DOWN   |
| 450 0419 EB     |                   | XCHG |        | "FROM" POINTER TO D&E      |
| 451 041A 2A5502 |                   | LHLD | NXTCHR |                            |
| 452 041D CD3304 |                   | CALL | NEGHL  |                            |
| 453 0420 19     |                   | DAD  | D      | FORM COUNT TO BE MOVED     |
| 454 0421 44     |                   | MOV  | B,H    |                            |
| 455 0422 4D     |                   | MOV  | C,L    |                            |
| 456 0423 2A3D02 |                   | LHLD | BADR1  | "TO" ADDRESS TO H&L        |
| 457 0426 1A     | DL2               | LDAX | D      | GET CHARACTER              |
| 458 0427 77     |                   | MOV  | M,A    | MOVE IT DOWN               |
| 459 0428 13     |                   | INX  | D      | INCREMENT "FROM" POINTER   |
| 460 0429 23     |                   | INX  | H      | INCREMENT "TO" POINTER     |
| 461 042A 03     |                   | INX  | B      | INCREMENT COUNT            |
| 462 042B 78     |                   | MOV  | A,B    |                            |
| 463 042C B1     |                   | ORA  | C      | TEST FOR ZERO COUNT        |
| 464 042D C22604 |                   | JNZ  | DL2    |                            |
| 465 0430 C30F04 |                   | JMP  | DL0    |                            |

```

466
467
468
469 0433 7C
470 0434 2F
471 0435 67
472 0436 7D
473 0437 2F
474 0438 6F
475 0439 23
476 043A C9
477
478
479
480 043B CDD001
481 043E FERE
482 0440 C2E002
483 0443 CD8901
484 0446 224F02
485 0449 EB
486 044A 2A3B02
487 044D CD3304
488 0450 19
489 0451 2B
490 0452 7C
491 0453 B7
492 0454 F2E002
493
494 0457 EB
495 0458 CDAB02
496 045B 13
497 045C EB
498 045D 224D02
499 0460 216202
500 0463 3E48
501 0465 325E02
502 0468 CD8901
503
504 046B CD7D01
505 046E FE8A
506 0470 CA6B04
507 0473 FEFF
508 0475 CA6B04
509 0478 FERE
510 047A CA1901
511 047D FE8D
512 047F CA6B04
513 0482 FEDF
514 0484 CA6B04
515 0487 C39C04
516 048A CD7D01
517 048D FE8A
518 048F CA8A04
519 0492 FEFF
520 0494 CA8A04
521 0497 FEDF
522 0499 CAB304
523 049C 77

*
* NEGATE THE 16 BIT NUMBER IN H&L
*
NEGHL    MOV    A,H
          CMA
          MOV    H,A
          MOV    A,L
          CMA
          MOV    L,A
          INX    H
          RET

*
* ADD FUNCTION
*
ADD      CALL    PFCH      GET LINE NUMBER
          CPI     ' '      PROPER TERMINATOR?

          JNE     ERR
          CALL    CRLF
          SHLD    FIRST
NXT      XCHG
          LHL     LNCNT     GET LINECOUNT
          CALL    NEGHL
          DAD     D         FORM LINE# - LINECOUNT
          DCX     H         -1
          MOV     A,H
          ORA     A         SET FLAGS
          JP      ERR      IF THIS JUMP TAKES PLACE
                           THE LINE # WAS > LINECOUNT
                           LINE # TO H-L
                           GET LINE ADDRESSES
                           FORM ADDRESS OF BEGINNING OF

          XCHG
          CALL    FBA
          INX     D
          XCHG
          SHLD    EOLADR    SAVE
          LXI     H,KB
          MVI     A,72
          STA     ACNT      GET CHARACTER COUNT/LINE
          CALL    CRLF
          ADI     EQU AD1A
          ADDI    EQU AD1A
          ADD2    CALL    RKB
          CPI     :8A
          JEQ     ADD2      LINE FEED?
          CPI     :FF
          JEQ     ADD2      RUBOUT?
          CPI     ' '
          JEQ     MON2      RETURN TO MONITOR?
          CPI     :8D
          JEQ     ADD1
          CPI     :DF
          JEQ     ADD1      BACK ARROW?
          JMP     ADD4
          CALL    RKB
          CPI     :8A
          JEQ     ADD3
          CPI     :FF
          JEQ     ADD3
          CPI     :DF
          JEQ     ADD5      BACKSLASH
          MOV     M,A       CHARACTER TO BUFFER

```



```

524 049D 23          INX    H
525 049E FE8D        CPI    :8D
526 04A0 CACA04      JEQ    ADD6
527 04A3 3A5E02      LDA    ACNT
528 04A6 3D          DCR    A          END OF LINE?
529 04A7 325E02      STA    ACNT
530 04AA C28A04      JNZ    ADD3
531 04AD CD8901      CALL   CRLF
532 04B0 C3CA04      JMP    ADD6
533 04B3 2B          ADD5   DCX    H
534 04B4 EB          XCHG
535 04B5 219EFD      LXI    H,-KB
536 04B8 19          DAD    D
537 04B9 7C          MOV    A,H
538 04BA B5          ORA    L
539 04BB CA6004      JZ     ADD0
540 04BE EB          XCHG
541 04BF CD7D01      CALL   RKB
542 04C2 FE8D        CPI    :8D
543 04C4 CA6004      JEQ    ADD0
544 04C7 C38D04      JMP    ADD3A
545 04CA EB          ADD6   XCHG
546 04CB 219EFD      LXI    H,-KB
547 04CE 19          DAD    D
548 04CF 4D          MOV    C,L
549 04D0 44          MOV    B,H
550 04D1 226002      SHLD   DCNT
551 04D4 2A5502      ADD7   LHL   NXTCHR  GET OLD NXTCHR
552 04D7 09          DAD    B          FORM NEW NXTCHR
553 04D8 EB          XCHG          SAVE
554 04D9 3A4902      LDA    EOB          CHECK FOR BUFFER OVERFLOW
555 04DC 93          SUB    E
556 04DD 3A4A02      LDA    EOB+1
557 04E0 9A          SBB    D
558 04E1 DA2905      JC     BFL          BUFFER OVERFLOW
559 04E4 00          NOP
560 04E5 00          NOP
561
562
563
564 04E6 2A5502      *
565 04E9 EB          * COMPUTE MOVE ADDRESSES
566 04EA 225502      *
567 04ED 44          LHL   NXTCHR
568 04EE 4D          XCHG
569 04EF 2A4D02      SHLD   NXTCHR  UPDATE NXTCHR
570 04F2 CD3304      MOV    B,H
571 04F5 19          MOV    C,L
572 04F6 7C          LHL   EOLADR  CHECK FOR LAST LINE
573 04F7 B5          CALL   NEGHL
574 04F8 CA0705      DAD    D
575 04FB 1B          MOV    A,H
576 04FC 0B          ORA    L
577 04FD 1A          JZ     ADD10
578 04FE 02          DCX    D
579 04FF 0B          DCX    B
580 0500 1B          DCX    B
581 0501 2B          ADD9   DCX    D
                    DCX    H

```

|                   |                 |      |               |                  |
|-------------------|-----------------|------|---------------|------------------|
| 582 0502 7C       |                 | MOV  | A,H           |                  |
| 583 0503 B5       |                 | ORA  | L             |                  |
| 584 0504 C2FD04   |                 | JNZ  | ADD9          |                  |
| 585 0507 016202   | ADD10           | LXI  | B,KB          |                  |
| 586 050A 2A4D02   |                 | LHLD | EOLADR        |                  |
| 587 050D EB       |                 | XCHG |               |                  |
| 588 050E 2A6002   |                 | LHLD | DCNT          |                  |
| 589 0511 0A       | ADD11           | LDAX | B             | MOVE IN NEW LINE |
| 590 0512 12       |                 | STAX | D             |                  |
| 591 0513 03       |                 | INX  | B             |                  |
| 592 0514 13       |                 | INX  | D             |                  |
| 593 0515 2B       |                 | DCX  | H             |                  |
| 594 0516 7C       |                 | MOV  | A,H           |                  |
| 595 0517 B5       |                 | ORA  | L             |                  |
| 596 0518 C21105   |                 | JNZ  | ADD11         |                  |
| 597 051B 2A3B02   |                 | LHLD | LNCNT         |                  |
| 598 051E 23       |                 | INX  | H             |                  |
| 599 051F 223B02   |                 | SHLD | LNCNT         |                  |
| 600 0522 2A4F02   |                 | LHLD | FIRST         |                  |
| 601 0525 23       |                 | INX  | H             |                  |
| 602 0526 C34604   |                 | JMP  | NXT           |                  |
| 603 0529 213205   | BFL             | LXI  | H,BFLA        |                  |
| 604 052C CD9901   |                 | CALL | PRT           |                  |
| 605 052F C31901   |                 | JMP  | MON2          |                  |
| 606 0532 C2D5C6C6 | BFLA            | ASC  | 'BUFFER FULL' |                  |
| 607 053D 8D       |                 | DATA | :8D           |                  |
| 608               | *               |      |               |                  |
| 609               | * READ FUNCTION |      |               |                  |
| 610               | *               |      |               |                  |
| 611 053E CD7D01   | READ            | CALL | RKB           | GET PERIOD       |
| 612 0541 FEAE     |                 | CPI  | ' '           |                  |
| 613 0543 C2E002   |                 | JNE  | ERR           |                  |
| 614 0546 CD8301   |                 | CALL | REWIND        |                  |
| 615 0549 AF       |                 | ZAR  |               |                  |
| 616 054A 325F02   |                 | STA  | NULLS         |                  |
| 617 054D CD5F01   |                 | CALL | RDRINT        |                  |
| 618 0550 CDC505   | R2              | CALL | RTR           |                  |
| 619 0553 DA1901   |                 | JC   | MON2          |                  |
| 620 0556 216202   |                 | LXI  | H,KB          |                  |
| 621 0559 EB       |                 | XCHG |               |                  |
| 622 055A 2A5502   |                 | LHLD | NXTCHR        |                  |
| 623 055D 3A4902   | R6              | LDA  | EOB           |                  |
| 624 0560 95       |                 | SUB  | L             |                  |
| 625 0561 4F       |                 | MOV  | C,A           |                  |
| 626 0562 3A4A02   |                 | LDA  | EOB+1         |                  |
| 627 0565 9C       |                 | SBB  | H             |                  |
| 628 0566 B1       |                 | ORA  | C             |                  |
| 629 0567 CA2905   |                 | JZ   | BFL           |                  |
| 630 056A 1A       |                 | LDAX | D             |                  |
| 631 056B 77       |                 | MOV  | M,A           |                  |
| 632 056C 23       |                 | INX  | H             |                  |
| 633 056D 13       |                 | INX  | D             |                  |
| 634 056E 05       |                 | DCR  | B             |                  |
| 635 056F C25D05   |                 | JNZ  | R6            |                  |
| 636 0572 225502   |                 | SHLD | NXTCHR        |                  |
| 637 0575 2A3B02   |                 | LHLD | LNCNT         |                  |
| 638 0578 23       |                 | INX  | H             |                  |
| 639 0579 223B02   |                 | SHLD | LNCNT         |                  |

```

640 057C CD6801      CALL DTCTCC
641 057F DA1901      JC  MON2
642 0582 C35005      JMP  R2
643 0585              RES  20
644
645                  *
646                  * READ CHARACTER FROM READER
647 0599 CD7101      RCH  CALL  GTPCHR   GET FRAME FROM PTR
648 059C F680        ORI   :80
649 059E FEFF        CPI   :FF
650 05A0 CA9905      JEQ   RCH
651 05A3 FE8A        CPI   :8A
652 05A5 CA9905      JEQ   RCH
653 05A8 FE80        CPI   :80
654 05AA CAB605      JEQ   RCH2
655 05AD F5          PUSH  PSW
656 05AE 3E0A        MVI   A,10
657 05B0 325F02      STA   NULLS
658 05B3 F1          POP   PSW
659 05B4 B7          RSC
660 05B5 C9          RET
661 05B6 3A5F02      RCH2 LDA   NULLS
662 05B9 3D          DCR   A
663 05BA 325F02      STA   NULLS
664 05BD C29905      JNZ   RCH
665 05C0 CD6E01      CALL  RDRSTP
666 05C3 37          STC
667 05C4 C9          RET
668
669                  *
670                  * READ PAPER TAPE RECORD FROM READER
671 05C5 216202      RTR  LXI   H,KB
672 05C8 0600        MVI   B,0
673 05CA CD6B01      CALL  RDRSTT
674 05CD CD9905      CALL  RCH
675 05D0 D8          RC
676 05D1 FEA1        CPI   '/'      CHECK FOR END OF TAPE
677 05D3 C2FA05      JNE   R3A
678 05D6 CD7101      RTRB CALL  GTPCHR
679 05D9 CD7101      CALL  GTPCHR
680 05DC CD7101      CALL  GTPCHR
681 05DF CD6E01      CALL  RDRSTP
682 05E2 CD7D01      RTRCZ CALL  RKB
683 05E5 FE83        CPI   :83
684 05E7 CA1901      JEQ   MON2
685 05EA FEA0        CPI   '/'
686 05EC C2E205      JNE   RTRCZ
687 05EF AF          ZAR
688 05F0 325F02      STA   NULLS
689 05F3 C3C505      JMP   RTR
690 05F6 CD9905      R3   CALL  RCH
691 05F9 D8          RC
692 05FA 77          R3A  MOV   M,A
693 05FB 23          INX   H
694 05FC 04          INR   B
695 05FD FE8D        CPI   :8D
696 05FF CA1606      JEQ   R5
697 0602 78          MOV   A,B

```

|                 |        |                                            |         |                      |
|-----------------|--------|--------------------------------------------|---------|----------------------|
| 698 0603 FE48   |        | CPI                                        | 72      |                      |
| 699 0605 C2F605 |        | JNE                                        | R3      |                      |
| 700 0608 3E8D   |        | MVI                                        | A, :8D  |                      |
| 701 060A 32AA02 |        | STA                                        | KB+72   |                      |
| 702 060D CD9905 | R4     | CALL                                       | RCH     |                      |
| 703 0610 D8     |        | RC                                         |         |                      |
| 704 0611 FE8D   |        | CPI                                        | :8D     |                      |
| 705 0613 C20D06 |        | JNE                                        | R4      |                      |
| 706 0616 78     | R5     | MOV                                        | A,B     |                      |
| 707 0617 FE01   |        | CPI                                        | 1       |                      |
| 708 0619 CAC505 |        | JEQ                                        | RTR     |                      |
| 709 061C CD6E01 |        | CALL                                       | RDRSTP  |                      |
| 710 061F B7     |        | RSC                                        |         |                      |
| 711 0620 C9     |        | RET                                        |         |                      |
| 712             |        | *                                          |         |                      |
| 713             |        | * FOLLOWING ROUTINE FORMATS THE RECORD     |         |                      |
| 714             |        | * WHOSE ADDRESS IS IN H&L ON ENTRY. RESULT |         |                      |
| 715             |        | * IS LEFT IN HEX, HEX+71                   |         |                      |
| 716             |        | *                                          |         |                      |
| 717 0621 AF     | FORMAT | ZAR                                        |         |                      |
| 718 0622 328607 |        | STA                                        | RTNFLG  | CLEAR FLAG           |
| 719 0625 1648   |        | MVI                                        | D,72    |                      |
| 720 0627 018907 |        | LXI                                        | B,HEX   |                      |
| 721 062A 3EA0   |        | MVI                                        | A,' '   |                      |
| 722 062C 02     | FRM1   | STAX                                       | B       | CLEAR WORK AREA      |
| 723 062D 03     |        | INX                                        | B       |                      |
| 724 062E 15     |        | DCR                                        | D       |                      |
| 725 062F C22C06 |        | JNZ                                        | FRM1    |                      |
| 726             |        | *                                          |         |                      |
| 727 0632 3EFF   |        | MVI                                        | A,-1    | SET NULL INSTRUCTION |
| 728 0634 328507 |        | STA                                        | SKEL    | SKELETON             |
| 729 0637 328407 |        | STA                                        | CLASS   | AND CLASS            |
| 730             |        | *                                          |         |                      |
| 731 063A 7E     |        | MOV                                        | A,M     | GET FIRST CHARACTER  |
| 732 063B 328307 |        | STA                                        | CHI     | SAVE FIRST CHARACTER |
| 733 063E FEA0   |        | CPI                                        | '*'     | COMMENT LINE?        |
| 734 0640 CA6E07 |        | JEQ                                        | XCLINE  | YES                  |
| 735 0643 FEA0   |        | CPI                                        | ' '     | NO, LABEL PRESENT?   |
| 736 0645 CA7106 |        | JEQ                                        | XOP     | NO                   |
| 737             |        | *                                          |         |                      |
| 738 0648 019D07 |        | LXI                                        | B,LABEL | TRANSMIT LABEL       |
| 739 064B 1606   |        | MVI                                        | D,6     |                      |
| 740 064D 7E     | FRM2   | MOV                                        | A,M     |                      |
| 741 064E FEA0   |        | CPI                                        | ' '     |                      |
| 742 0650 CA7106 |        | JEQ                                        | XOP     | END OF LABEL         |
| 743 0653 02     |        | STAX                                       | B       |                      |
| 744 0654 23     |        | INX                                        | H       |                      |
| 745 0655 03     |        | INX                                        | B       |                      |
| 746 0656 FE8D   |        | CPI                                        | :8D     |                      |
| 747 0658 CA5207 |        | JEQ                                        | ENDFMT  |                      |
| 748 065B 15     |        | DCR                                        | D       |                      |
| 749 065C C24D06 |        | JNZ                                        | FRM2    |                      |
| 750             |        | *                                          |         |                      |
| 751 065F 7E     | TR     | MOV                                        | A,M     | WASTE EXCESS LABEL   |
| 752 0660 23     |        | INX                                        | H       |                      |
| 753 0661 FEA0   |        | CPI                                        | ' '     |                      |
| 754 0663 CA7106 |        | JEQ                                        | XOP     |                      |
| 755 0666 FE8D   |        | CPI                                        | :8D     |                      |

|                 |      |         |          |
|-----------------|------|---------|----------|
| 756 0668 C25F06 | JNE  | TR      |          |
| 757 066B 32A307 | STA  | LABEL+6 | SET C.R. |
| 758 066E C35207 | JMP  | ENDFMT  |          |
| 759             | *    |         |          |
| 760 0671 7E     | XOP  | MOV     | A,M      |
| 761 0672 FEA0   |      | CPI     | ' '      |
| 762 0674 C27B06 |      | JNE     | XOP1     |
| 763 0677 23     |      | INX     | H        |
| 764 0678 C37106 |      | JMP     | XOP      |
| 765 067B 1604   | XOP1 | MVI     | D,4      |
| 766 067D 01A507 |      | LXI     | B,MNEM   |
| 767 0680 7E     | FRM3 | MOV     | A,M      |
| 768 0681 FEA0   |      | CPI     | ' '      |
| 769 0683 CAA706 |      | JEQ     | LUOP     |
| 770 0686 FE8D   |      | CPI     | :8D      |
| 771 0688 CAA106 |      | JEQ     | FRM5     |
| 772 068B 02     |      | STAX    | B        |
| 773 068C 23     |      | INX     | H        |
| 774 068D 03     |      | INX     | B        |
| 775 068E 15     |      | DCR     | D        |
| 776 068F C28006 |      | JNZ     | FRM3     |
| 777 0692 7E     | FRM4 | MOV     | A,M      |
| 778 0693 23     |      | INX     | H        |
| 779 0694 FE8D   |      | CPI     | :8D      |
| 780 0696 CAA106 |      | JEQ     | FRM5     |
| 781 0699 FEA0   |      | CPI     | ' '      |
| 782 069B CAA706 |      | JEQ     | LUOP     |
| 783 069E C39206 |      | JMP     | FRM4     |
| 784 06A1 328607 | FRM5 | STA     | RTNFLG   |
| 785 06A4 32A907 |      | STA     | MNEM+4   |
| 786             | *    |         |          |
| 787 06A7 228707 | LUOP | SHLD    | HSAVE    |
| 788 06AA 2AA707 |      | LHLD    | MNEM+2   |
| 789 06AD EB     |      | XCHG    |          |
| 790 06AE 2AA507 |      | LHLD    | MNEM     |
| 791 06B1 CDD207 |      | CALL    | OPSRC    |
| 792 06B4 228407 |      | SHLD    | CLASS    |
| 793 06B7 3A8607 |      | LDA     | RTNFLG   |
| 794 06BA B7     |      | ORA     | A        |
| 795 06BB C25207 |      | JNZ     | ENDFMT   |
| 796 06BE 55     |      | MOV     | D,L      |
| 797 06BF 2A8707 |      | LHLD    | HSAVE    |
| 798 06C2 7E     | FRM6 | MOV     | A,M      |
| 799 06C3 23     |      | INX     | H        |
| 800 06C4 FEA0   |      | CPI     | ' '      |
| 801 06C6 CAC206 |      | JEQ     | FRM6     |
| 802 06C9 FE8D   |      | CPI     | :8D      |
| 803 06CB C2D406 |      | JNE     | FRM7     |
| 804 06CE 32A907 |      | STA     | MNEM+4   |
| 805 06D1 C35207 |      | JMP     | ENDFMT   |
| 806             | *    |         |          |
| 807 06D4 2B     | FRM7 | DCX     | H        |
| 808 06D5 7A     |      | MOV     | A,D      |
| 809 06D6 110000 |      | LXI     | D,0      |
| 810 06D9 B7     |      | ORA     | A        |
| 811 06DA FA1A07 |      | JM      | FRM12    |
| 812 06DD 01AB07 |      | LXI     | B,OPNDF  |
| 813 06E0 C2EB06 |      | JNZ     | FRM8     |

CHECK FOR PRESENCE OF  
OPERAND



|                 |        |      |           |                       |
|-----------------|--------|------|-----------|-----------------------|
| 814 06E3 3A8507 |        | LDA  | SKEL      | CHECK FOR TITL        |
| 815 06E6 FE03   |        | CPI  | 3         |                       |
| 816 06E8 CA5907 |        | JEQ  | XMTTL     |                       |
| 817             |        |      |           | MOVE OPERAND          |
| 818 06EB 7E     | FRM8   | MOV  | A,M       |                       |
| 819 06EC FE8D   |        | CPI  | :8D       |                       |
| 820 06EE CA1607 |        | JEQ  | FRM11     |                       |
| 821 06F1 FEA7   |        | CPI  | :A7       | APOSTROPHE?           |
| 822 06F3 C2FE06 |        | JNE  | FRM9      |                       |
| 823 06F6 7B     |        | MOV  | A,E       |                       |
| 824 06F7 2F     |        | CMA  |           |                       |
| 825 06F8 5F     |        | MOV  | E,A       | SET APOSTROPHE FLAG   |
| 826 06F9 3EA7   |        | MVI  | A,:A7     |                       |
| 827 06FB C30A07 |        | JMP  | FRM10     |                       |
| 828 06FE FEA0   | FRM9   | CPI  | ''        |                       |
| 829 0700 C20A07 |        | JNE  | FRM10     |                       |
| 830 0703 7B     |        | MOV  | A,E       |                       |
| 831 0704 B7     |        | ORA  | A         |                       |
| 832 0705 CA1A07 |        | JZ   | FRM12     |                       |
| 833 0708 3EA0   |        | MVI  | A, ''     |                       |
| 834 070A 02     | FRM10  | STAX | B         |                       |
| 835 070B 23     |        | INX  | H         |                       |
| 836 070C 03     |        | INX  | B         |                       |
| 837 070D 14     |        | INR  | D         |                       |
| 838 070E 7A     |        | MOV  | A,D       |                       |
| 839 070F FE26   |        | CPI  | 38        |                       |
| 840 0711 C2EB06 |        | JNE  | FRM8      |                       |
| 841 0714 3E8D   |        | MVI  | A,:8D     |                       |
| 842 0716 02     | FRM11  | STAX | B         |                       |
| 843 0717 C35207 |        | JMP  | ENDFMT    |                       |
| 844             | *      |      |           |                       |
| 845             |        |      |           | MOVE COMMENT FIELD    |
| 846 071A 7E     | FRM12  | MOV  | A,M       |                       |
| 847 071B FE8D   |        | CPI  | :8D       |                       |
| 848 071D CA1607 |        | JEQ  | FRM11     |                       |
| 849 0720 FEA0   |        | CPI  | ''        |                       |
| 850 0722 C22907 |        | JNE  | FRM12A    |                       |
| 851 0725 23     |        | INX  | H         |                       |
| 852 0726 C31A07 |        | JMP  | FRM12     |                       |
| 853 0729 7A     | FRM12A | MOV  | A,D       |                       |
| 854 072A 010000 |        | LXI  | B,0       |                       |
| 855 072D FE0A   |        | CPI  | 10        |                       |
| 856 072F FA3507 |        | JC   | XCF       |                       |
| 857 0732 DE09   |        | SBI  | 9         | COMPUTE COMMENT START |
| 858 0734 4F     |        | MOV  | C,A       | ADDRESS               |
| 859 0735 EB     | XCF    | XCHG |           |                       |
| 860 0736 21B507 |        | LXI  | H,CFLD    |                       |
| 861 0739 09     |        | DAD  | B         |                       |
| 862 073A 012FF8 |        | LXI  | B,-HEX-72 |                       |
| 863 073D 228707 |        | SHLD | HSAVE     |                       |
| 864 0740 09     |        | DAD  | B         |                       |
| 865 0741 7C     |        | MOV  | A,H       |                       |
| 866 0742 B5     |        | ORA  | L         |                       |
| 867 0743 F25207 |        | JP   | ENDFMT    | NO ROOM FOR COMMENT   |
| 868 0746 45     |        | MOV  | B,L       |                       |
| 869 0747 2A8707 |        | LHLD | HSAVE     |                       |
| 870 074A 1A     | FRM13  | LDAX | D         |                       |
| 871 074B 77     |        | MOV  | M,A       | MOVE COMMENT          |

```

872 074C 13          INX  D
873 074D 23          INX  H
874 074E 04          INR  B
875 074F C24A07      JNZ  FRM13
876 0752 2A8407      ENDFMT LHL  CLASS
877 0755 3A8307      LDA  CH1
878 0758 C9          RET
879
880                  * TRANSMIT TITLE LINE
881                  *
882 0759 1626          XMTTL  MVI  D,38
883 075B 01AB07        LXI  B,OPNDF
884 075E 7E          XMTTL2  MOV  A,M
885 075F 02          STAX  B
886 0760 23          INX  H
887 0761 03          INX  B
888 0762 FE8D          CPI  :8D
889 0764 CA6B07        JEQ  XMTTLE
890 0767 15          DCR  D
891 0768 C25E07        JNZ  XMTTL2
892 076B C35207        XMTTLE JMP  ENDFMT
893
894                  * TRANSMIT COMMENTS LINE
895                  *
896 076E 1634          XCLINE  MVI  D,52
897 0770 019D07        LXI  B,LABEL
898 0773 7E          XCLI  MOV  A,M
899 0774 02          STAX  B
900 0775 23          INX  H
901 0776 03          INX  B
902 0777 FE8D          CPI  :8D
903 0779 CA5207        JEQ  ENDFMT
904 077C 15          DCR  D
905 077D C27307        JNZ  XCLI
906 0780 C35207        JMP  ENDFMT
907
908 0783 00          *
909 0784 00          CHI  ***
910 0785 00          CLASS ***
911 0786 00          SKEL ***
912 0787 0000        RTNFLG ***
913 0789            HSAVE DBL  0
914 079D            HEX  RES  20
915 07A5            LABEL RES  8
916 07AB            MNEM  RES  6
917 07B5            OPNDF RES  10
918 07D1 8D          CFLD  RES  28
919                  DATA  :8D
920
921                  *****
922                  *
923                  * SEARCH Z80 OPCODE TABLE FOR INSTRUCTION
924                  * OR PSEUDO-OP. ENTER WITH MNEMONIC IN H-L,
925                  * AND D-E. RETURN WITH INSTRUCTION CLASS IN
926                  * L AND INSTRUCTION SKELETON IN H, CARRY OFF.
927                  * IF MNEMONIC NOT FOUND CARRY RETURNS ON.
928                  *
929                  * IN THIS TABLE THE HIGH BIT OF THE INSTRUCTION
930                  * CLASS WORD IS SET TO ONE IF THE INSTRUCTION
931                  * DOES NOT REQUIRE AN OPERAND. THIS IS FOR THE

```

```

930          * USE OF THE FORMAT ROUTINE.
931          *
932          *
933 07D2 220E08  OPSRC  SHLD  OP          ENTRY, SAVE MNEMONIC
934 07D5 EB      XCHG
935 07D6 221008      SHLD  OP+2
936 07D9 016201      LXI   B, TSIZE
937 07DC 211208      LXI   H, ITBL
938 07DF 110E08  OPSRCI LXI   D, OP
939 07E2 C5      PUSH  B
940 07E3 CDFE07      CALL  OPCMPR
941 07E6 C1      POP   B
942 07E7 B7      ORA   A          SET FLAGS
943 07E8 CAF807      JZ    EUREKA
944 07EB 23      INX   H
945 07EC 23      INX   H
946 07ED 0B      DCX   B
947 07EE 78      MOV   A, B
948 07EF B1      ORA   C
949 07F0 C2DF07      JNZ  OPSRCI
950 07F3 21FFFF      LXI   H, -1
951 07F6 37      STC
952 07F7 C9      RET
953 07F8 56      EUREKA MOV   D, M
954 07F9 23      INX   H
955 07FA 5E      MOV   E, M
956 07FB EB      XCHG
957 07FC B7      RSC
958 07FD C9      RET
959          * COMPARE 4 CHAR MNEMONIC
960 07FE 0604  OPCMPR MVI   B, 4
961 0800 0E00      MVI   C, 0
962 0802 1A      OPCMPI LDAX  D
963 0803 96      SUB   M
964 0804 B1      ORA   C
965 0805 4F      MOV   C, A
966 0806 13      INX   D
967 0807 23      INX   H
968 0808 05      DCR   B
969 0809 C20208      JNZ  OPCMPI
970 080C 79      MOV   A, C
971 080D C9      RET
972 080E      OP      RES   4
973          TSIZE  EQU   354          OPCODE TABLE SIZE
974          *
975          * CLASS 0 - PSEUDO OPS
976          *
977 0812 CFD2C7A0  ITBL  ASC   'ORG '
978 0816 0000      DBL   0
979 0818 C5CEC4A0  ASC   'END '
980 081C 0180      DBL   '8001'
981 081E C1D2D3A0  ASC   'ARS '
982 0822 0280      DBL   '8002'
983 0824 D4C9D4CC  ASC   'TITL'
984 0828 0300      DBL   3
985 082A D5CECCA0  ASC   'UNL '
986 082E 0480      DBL   '8004'
987 0830 CCD3D4A0  ASC   'LST '

```

```

988 0834 0580      DBL :8005
989 0836 C4C1D4C1  ASC 'DATA'
990 083A 0600      DBL 6
991 083C C5D1D5A0  ASC 'EQU '
992 0840 0700      DBL 7
993 0842 C1D3C3A0  ASC 'ASC '
994 0846 0800      DBL 8
995 0848 D2C5D3A0  ASC 'RES '
996 084C 0900      DBL 9
997 084E C2D3D3A0  ASC 'BSS '
998 0852 0900      DBL 9
999 0854 D2C5CDA0  ASC 'REM '
000 0858 0A80      DBL :800A
001 085A C4C2CCA0  ASC 'DBL '
002 085E 0B00      DBL 11
003
004
005
006
007
008
009
010 0860 C1C4C4A0  ASC 'ADD '
011 0864 8001      DBL :180
012 0866 C1C4C3A0  ASC 'ADC '
013 086A 8801      DBL :188
014 086C D3D5C2A0  ASC 'SUB '
015 0870 9001      DBL :190
016 0872 D3C2C2A0  ASC 'SBB '
017 0876 9801      DBL :198
018 0878 C9CED2A0  ASC 'INR '
019 087C 0401      DBL :104
020 087E C4C3D2A0  ASC 'DCR '
021 0882 0501      DBL :105
022 0884 C1CEC1A0  ASC 'ANA '
023 0888 A001      DBL :1A0
024 088A D8D2C1A0  ASC 'XRA '
025 088E A801      DBL :1A8
026 0890 CFD2C1A0  ASC 'ORA '
027 0894 B001      DBL :1B0
028 0896 C3CDD0A0  ASC 'CMP '
029 089A B801      DBL :1B8
030
031
032
033 089C CDCFD6A0  ASC 'MOV '
034 08A0 4002      DBL :240
035
036
037
038 08A2 CDD6C9A0  ASC 'MVI '
039 08A6 0603      DBL :306
040
041
042
043 08A8 CCC4C1A0  ASC 'LDA '
044 08AC 3A04      DBL :43A
045 08AE D3D4C1A0  ASC 'STA '

```

\* HOW ABOUT ADDING TPL, QUAD AND FPC PSEUDO OPS  
 \* FOR TRIPLE, QUADRUPLE AND FLOATING POINT  
 \* CONSTANTS.  
 \*  
 \* CLASS 1 - SINGLE REGISTER  
 \*  
 \* CLASS 2 - 2 SINGLE REGISTERS  
 \*  
 \* CLASS 3 - SINGLE REG. + DATA WORD  
 \*  
 \* CLASS 4 - 16 BIT OPERAND  
 \*

|                    |            |
|--------------------|------------|
| 046 08B2 3204      | DBL :432   |
| 047 08B4 CCC8CCC4  | ASC 'LHLD' |
| 048 08B8 2A04      | DBL :42A   |
| 049 08BA D3C8CCC4  | ASC 'SHLD' |
| 050 08BE 2204      | DBL :422   |
| 051 08C0 CACDD0A0  | ASC 'JMP'  |
| 052 08C4 C304      | DBL :4C3   |
| 053 08C6 CADAA0A0  | ASC 'JZ'   |
| 054 08CA CA04      | DBL :4CA   |
| 055 08CC CAC5D1A0  | ASC 'JEQ'  |
| 056 08D0 CA04      | DBL :4CA   |
| 057 08D2 CACEDAA0  | ASC 'JNZ'  |
| 058 08D6 C204      | DBL :4C2   |
| 059 08D8 CACEC5A0  | ASC 'JNE'  |
| 060 08DC C204      | DBL :4C2   |
| 061 08DE CAC3A0A0  | ASC 'JC'   |
| 062 08E2 DA04      | DBL :4DA   |
| 063 08E4 CAC1CCA0  | ASC 'JAL'  |
| 064 08E8 DA04      | DBL :4DA   |
| 065 08EA CACCD3A0  | ASC 'JLS'  |
| 066 08EE DA04      | DBL :4DA   |
| 067 08F0 CACEC3A0  | ASC 'JNC'  |
| 068 08F4 D204      | DBL :4D2   |
| 069 08F6 CAC7C5A0  | ASC 'JGE'  |
| 070 08FA D204      | DBL :4D2   |
| 071 08FC CAD0CFA0  | ASC 'JPO'  |
| 072 0900 E204      | DBL :4E2   |
| 073 0902 CACED6A0  | ASC 'JNV'  |
| 074 0906 E204      | DBL :4E2   |
| 075 0908 CAD0C5A0  | ASC 'JPE'  |
| 076 090C EA04      | DBL :4EA   |
| 077 090E CAD6A0A0  | ASC 'JV'   |
| 078 0912 EA04      | DBL :4EA   |
| 079 0914 CAD0A0A0  | ASC 'JP'   |
| 080 0918 F204      | DBL :4F2   |
| 081 091A CACDA0A0  | ASC 'JM'   |
| 082 091E FA04      | DBL :4FA   |
| 083 0920 CABEBDA0  | ASC 'J>='  |
| 084 0924 F204      | DBL :4F2   |
| 085 0926 CABCRA0A0 | ASC 'JK'   |
| 086 092A FA04      | DBL :4FA   |
| 087 092C C3C1CCCC  | ASC 'CALL' |
| 088 0930 CD04      | DBL :4CD   |
| 089 0932 C3CEDAA0  | ASC 'CNZ'  |
| 090 0936 C404      | DBL :4C4   |
| 091 0938 C3CEC5A0  | ASC 'CNE'  |
| 092 093C C404      | DBL :4C4   |
| 093 093E C3DAA0A0  | ASC 'CZ'   |
| 094 0942 CC04      | DBL :4CC   |
| 095 0944 C3C5A0A0  | ASC 'CE'   |
| 096 0948 CC04      | DBL :4CC   |
| 097 094A C3CEC3A0  | ASC 'CNC'  |
| 098 094E D404      | DBL :4D4   |
| 099 0950 C3CCD3A0  | ASC 'CLS'  |
| 100 0954 D404      | DBL :4D4   |
| 101 0956 C3C3A0A0  | ASC 'CC'   |
| 102 095A DC04      | DBL :4DC   |
| 103 095C C3C7C5A0  | ASC 'CGE'  |



|                   |                                        |       |                  |
|-------------------|----------------------------------------|-------|------------------|
| 104 0960 DC04     | DBL                                    | :4DC  |                  |
| 105 0962 C3D0CFA0 | ASC                                    | 'CPD  |                  |
| 106 0966 E404     | DBL                                    | :4E4  |                  |
| 107 0968 C3CED6A0 | ASC                                    | 'CNV  |                  |
| 108 096C E404     | DBL                                    | :4E4  |                  |
| 109 096E C3D0C5A0 | ASC                                    | 'CPE  |                  |
| 110 0972 EC04     | DBL                                    | :4EC  |                  |
| 111 0974 C3D6A0A0 | ASC                                    | 'CV   |                  |
| 112 0978 EC04     | DBL                                    | :4EC  |                  |
| 113 097A C3D0A0A0 | ASC                                    | 'CP   |                  |
| 114 097E F404     | DBL                                    | :4F4  |                  |
| 115 0980 C3BEBDA0 | ASC                                    | 'C>=  |                  |
| 116 0984 F404     | DBL                                    | :4F4  |                  |
| 117 0986 C3CDA0A0 | ASC                                    | 'CM   |                  |
| 118 098A FC04     | DBL                                    | :4FC  |                  |
| 119 098C C3BCA0A0 | ASC                                    | 'CC   |                  |
| 120 0990 FC04     | DBL                                    | :4FC  |                  |
| 121               |                                        |       |                  |
| 122               | *                                      |       |                  |
| 123               | * CLASS 5 - REG. PAIR + 16 BIT OPERAND |       |                  |
| 124 0992 CDD8C9A0 | ASC                                    | 'LXI  |                  |
| 125 0996 0105     | DBL                                    | :501  |                  |
| 126               | *                                      |       |                  |
| 127               | * CLASS 6 - REGISTER PAIR              |       |                  |
| 128               | *                                      |       |                  |
| 129 0998 CCC4C1D8 | ASC                                    | 'LDAX |                  |
| 130 099C 0A06     | DBL                                    | :60A  |                  |
| 131 099E D3D4C1D8 | ASC                                    | 'STAX |                  |
| 132 09A2 0206     | DBL                                    | :602  |                  |
| 133 09A4 C9CED8A0 | ASC                                    | 'INX  |                  |
| 134 09A8 0306     | DBL                                    | :603  |                  |
| 135 09AA C4C3D8A0 | ASC                                    | 'DCX  |                  |
| 136 09AE 0B06     | DBL                                    | :60B  |                  |
| 137 09B0 C4C1C4A0 | ASC                                    | 'DAD  |                  |
| 138 09B4 0906     | DBL                                    | :609  |                  |
| 139 09B6 D0D5D3C8 | ASC                                    | 'PUSH |                  |
| 140 09BA C506     | DBL                                    | :6C5  |                  |
| 141 09BC D0D3C8A0 | ASC                                    | 'PSH  |                  |
| 142 09C0 C506     | DBL                                    | :6C5  | SYNONYM FOR PUSH |
| 143 09C2 D0CFD0A0 | ASC                                    | 'POP  |                  |
| 144 09C6 C106     | DBL                                    | :6C1  |                  |
| 145               | *                                      |       |                  |
| 146               | * CLASS 7 - NO OPERAND                 |       |                  |
| 147               | *                                      |       |                  |
| 148 09C8 D8C3C8C7 | ASC                                    | 'XCHG |                  |
| 149 09CC EB87     | DBL                                    | :87EB |                  |
| 150 09CE C4C1C1A0 | ASC                                    | 'DAA  |                  |
| 151 09D2 2787     | DBL                                    | :8727 |                  |
| 152 09D4 D2CCC3A0 | ASC                                    | 'RLC  |                  |
| 153 09D8 0787     | DBL                                    | :8707 |                  |
| 154 09DA D2D2C3A0 | ASC                                    | 'RRC  |                  |
| 155 09DE 0F87     | DBL                                    | :870F |                  |
| 156 09E0 D2C1CCA0 | ASC                                    | 'RAL  |                  |
| 157 09E4 1787     | DBL                                    | :8717 |                  |
| 158 09E6 D2C1D2A0 | ASC                                    | 'RAR  |                  |
| 159 09EA 1F87     | DBL                                    | :871F |                  |
| 160 09EC C3CDC1A0 | ASC                                    | 'CMA  |                  |
| 161 09F0 2F87     | DBL                                    | :872F |                  |

|                   |     |        |
|-------------------|-----|--------|
| 162 09F2 C3CDC3A0 | ASC | 'CMC ' |
| 163 09F6 3F87     | DBL | :873F  |
| 164 09F8 D3D4C3A0 | ASC | 'STC ' |
| 165 09FC 3787     | DBL | :8737  |
| 166 09FE D2D3C3A0 | ASC | 'RSC ' |
| 167 0A02 B787     | DBL | :87B7  |
| 168 0A04 DAC1D2A0 | ASC | 'ZAR ' |
| 169 0A08 AF87     | DBL | :87AF  |
| 170 0A0A D2C5D4A0 | ASC | 'RET ' |
| 171 0A0E C987     | DBL | :87C9  |
| 172 0A10 D2CEDAA0 | ASC | 'RNZ ' |
| 173 0A14 C087     | DBL | :87C0  |
| 174 0A16 D2CEC5A0 | ASC | 'RNE ' |
| 175 0A1A C087     | DBL | :87C0  |
| 176 0A1C D2DAA0A0 | ASC | 'RZ '  |
| 177 0A20 C887     | DBL | :87C8  |
| 178 0A22 D2C5D1A0 | ASC | 'REQ ' |
| 179 0A26 C887     | DBL | :87C8  |
| 180 0A28 D2CEC3A0 | ASC | 'RNC ' |
| 181 0A2C D087     | DBL | :87D0  |
| 182 0A2E D2C7C5A0 | ASC | 'RGE ' |
| 183 0A32 D087     | DBL | :87D0  |
| 184 0A34 D2C3A0A0 | ASC | 'RC '  |
| 185 0A38 D887     | DBL | :87D8  |
| 186 0A3A D2CCD3A0 | ASC | 'RLS ' |
| 187 0A3E D887     | DBL | :87D8  |
| 188 0A40 D2D0CFA0 | ASC | 'RPO ' |
| 189 0A44 E087     | DBL | :87E0  |
| 190 0A46 D2CED6A0 | ASC | 'RNV ' |
| 191 0A4A E087     | DBL | :87E0  |
| 192 0A4C D2D0C5A0 | ASC | 'RPE ' |
| 193 0A50 E887     | DBL | :87E8  |
| 194 0A52 D2D6A0A0 | ASC | 'RV '  |
| 195 0A56 E887     | DBL | :87E8  |
| 196 0A58 D2D0A0A0 | ASC | 'RP '  |
| 197 0A5C F087     | DBL | :87F0  |
| 198 0A5E D2BEBDA0 | ASC | 'R>= ' |
| 199 0A62 F087     | DBL | :87F0  |
| 200 0A64 D2CDA0A0 | ASC | 'RM '  |
| 201 0A68 F887     | DBL | :87F8  |
| 202 0A6A D2BCA0A0 | ASC | 'RC '  |
| 203 0A6E F887     | DBL | :87F8  |
| 204 0A70 D0C3C8CC | ASC | 'PCHL' |
| 205 0A74 E987     | DBL | :87E9  |
| 206 0A76 D8D4C8CC | ASC | 'XTHL' |
| 207 0A7A E387     | DBL | :87E3  |
| 208 0A7C D3D0C8CC | ASC | 'SPHL' |
| 209 0A80 F987     | DBL | :87F9  |
| 210 0A82 C5C9A0A0 | ASC | 'EI '  |
| 211 0A86 FB87     | DBL | :87FB  |
| 212 0A88 C4C9A0A0 | ASC | 'DI '  |
| 213 0A8C F387     | DBL | :87F3  |
| 214 0A8E C8CCD4A0 | ASC | 'HLT ' |
| 215 0A92 7687     | DBL | :8776  |
| 216 0A94 CECFD0A0 | ASC | 'NOP ' |
| 217 0A98 0087     | DBL | :8700  |
| 218 0A9A AAAAAA00 | ASC | '*** ' |
| 219 0A9E 0087     | DBL | :8700  |

```

220
221      * CLASS 8 - ONE DATA WORD
222      *
223 0AA0 C1C4C9A0      ASC 'ADI '
224 0AA4 C608          DBL :806
225 0AA6 C1C3C9A0      ASC 'ACI '
226 0AAA CE08          DBL :8CE
227 0AAC D3D5C9A0      ASC 'SUI '
228 0AB0 D608          DBL :8D6
229 0AB2 D3C2C9A0      ASC 'SBI '
230 0AB6 DE08          DBL :8DE
231 0AB8 C1CEC9A0      ASC 'ANI '
232 0ABC E608          DBL :8E6
233 0ABE D8D2C9A0      ASC 'XRI '
234 0AC2 EE08          DBL :8EE
235 0AC4 CFD2C9A0      ASC 'ORI '
236 0AC8 F608          DBL :8F6
237 0ACA C3D0C9A0      ASC 'CPI '
238 0ACE FE08          DBL :8FE
239 0AD0 C9CEA0A0      ASC 'IN '
240 0AD4 DB08          DBL :8DB
241 0AD6 CFD5D4A0      ASC 'OUT '
242 0ADA D308          DBL :8D3
243
244      * CLASS 9 - TWO WORD PSEUDO OP
245      *
246 0ADC D4C3C1A0      ASC 'TCA '
247 0AE0 0189          DBL :8901
248 0AE2 D3D4CFD0      ASC 'STOP'
249 0AE6 0289          DBL :8902
250 0AE8 D7C1C9D4      ASC 'WAIT'
251 0AEC 0289          DBL :8902
252
253      * CLASS A - OPERAND 0-7
254      *
255 0AEE D2D3D4A0      ASC 'RST '
256 0AF2 C70A          DBL :AC7
257 0AF4 D4D2D0A0      ASC 'TRP '
258 0AF8 C70A          DBL :AC7
259
260      * Z80 INSTRUCTION TYPES
261      *
262      * CLASS B - 2 BYTES FIXED + DISPLACEMENT
263 0AFA CCC1D8A0      ASC 'LAX '
264 0AFE 7E0B          DBL :B7E
265 0B00 CCC2D8A0      ASC 'LBX '
266 0B04 460B          DBL :B46
267 0B06 CCC3D8A0      ASC 'LCX '
268 0B0A 4E0B          DBL :B4E
269 0B0C CCC4D8A0      ASC 'LDX '
270 0B10 560B          DBL :B56
271 0B12 CCC5D8A0      ASC 'LEX '
272 0B16 5E0B          DBL :B5E
273 0B18 CCC8D8A0      ASC 'LHX '
274 0B1C 660B          DBL :B66
275 0B1E CCCCD8A0      ASC 'LLX '
276 0B22 6E0B          DBL :B6E
277 0B24 D3C1D8A0      ASC 'SAX '

```

|                   |     |        |
|-------------------|-----|--------|
| 278 0B28 770B     | DBL | :B77   |
| 279 0B2A D3C2D8A0 | ASC | 'SBX'  |
| 280 0B2E 700B     | DBL | :B70   |
| 281 0B30 D3C3D8A0 | ASC | 'SCX'  |
| 282 0B34 710B     | DBL | :B71   |
| 283 0B36 D3C4D8A0 | ASC | 'SDX'  |
| 284 0B3A 720B     | DBL | :B72   |
| 285 0B3C D3C5D8A0 | ASC | 'SEX'  |
| 286 0B40 730B     | DBL | :B73   |
| 287 0B42 D3C8D8A0 | ASC | 'SHX'  |
| 288 0B46 740B     | DBL | :B74   |
| 289 0B48 D3CCD8A0 | ASC | 'SLX'  |
| 290 0B4C 750B     | DBL | :B75   |
| 291 0B4E CCC1D9A0 | ASC | 'LAY'  |
| 292 0B52 7E0B     | DBL | :B7E   |
| 293 0B54 CCC2D9A0 | ASC | 'LBY'  |
| 294 0B58 460B     | DBL | :B46   |
| 295 0B5A CCC3D9A0 | ASC | 'LCY'  |
| 296 0B5E 4E0B     | DBL | :B4E   |
| 297 0B60 CCC4D9A0 | ASC | 'LDY'  |
| 298 0B64 560B     | DBL | :B56   |
| 299 0B66 CCC5D9A0 | ASC | 'LEY'  |
| 300 0B6A 5E0B     | DBL | :B5E   |
| 301 0B6C CCC8D9A0 | ASC | 'LHY'  |
| 302 0B70 660B     | DBL | :B66   |
| 303 0B72 CCCC9A0  | ASC | 'LLY'  |
| 304 0B76 6E0B     | DBL | :B6E   |
| 305 0B78 D3C1D9A0 | ASC | 'SAY'  |
| 306 0B7C 770B     | DBL | :B77   |
| 307 0B7E D3C2D9A0 | ASC | 'SBY'  |
| 308 0B82 700B     | DBL | :B70   |
| 309 0B84 D3C3D9A0 | ASC | 'SCY'  |
| 310 0B88 710B     | DBL | :B71   |
| 311 0B8A D3C4D9A0 | ASC | 'SDY'  |
| 312 0B8E 720B     | DBL | :B72   |
| 313 0B90 D3C5D9A0 | ASC | 'SEY'  |
| 314 0B94 730B     | DBL | :B73   |
| 315 0B96 D3C8D9A0 | ASC | 'SHY'  |
| 316 0B9A 740B     | DBL | :B74   |
| 317 0B9C D3CCD9A0 | ASC | 'SLY'  |
| 318 0BA0 750B     | DBL | :B75   |
| 319 0BA2 C1C4C4D8 | ASC | 'ADDX' |
| 320 0BA6 860B     | DBL | :B86   |
| 321 0BA8 C1C4C3D8 | ASC | 'ADCX' |
| 322 0BAC 8E0B     | DBL | :B8E   |
| 323 0BAE D3D5C2D8 | ASC | 'SUBX' |
| 324 0BB2 960B     | DBL | :B96   |
| 325 0BB4 D3C2C3D8 | ASC | 'SBCX' |
| 326 0BB8 9E0B     | DBL | :B9E   |
| 327 0BBA C1CED8A0 | ASC | 'ANX'  |
| 328 0BBE A60B     | DBL | :BA6   |
| 329 0BC0 CFD2D8A0 | ASC | 'ORX'  |
| 330 0BC4 B60B     | DBL | :BB6   |
| 331 0BC6 D8D2D8A0 | ASC | 'XRX'  |
| 332 0BCA AE0B     | DBL | :BAE   |
| 333 0BCC C3CDD0D8 | ASC | 'CMPX' |
| 334 0BD0 BE0B     | DBL | :BBE   |
| 335 0BD2 C9CECD08 | ASC | 'INMX' |

```

336 0BD6 340B          DBL   :B34
337 0BD8 C4C3CDD8      ASC   'DCMX'
338 0BDC 350B          DBL   :B35
339 0BDE C1C4C4D9      ASC   'ADDY'
340 0BE2 860B          DBL   :B86
341 0BE4 C1C4C3D9      ASC   'ADCY'
342 0BE8 8E0B          DBL   :B8E
343 0BEA D3D5C2D9      ASC   'SUBY'
344 0BEE 960B          DBL   :B96
345 0BF0 D3C2C3D9      ASC   'SBCY'
346 0BF4 9E0B          DBL   :B9E
347 0BF6 C1CED9A0      ASC   'ANY '
348 0BFA A60B          DBL   :BA6
349 0BFC CFD2D9A0      ASC   'ORY '
350 0C00 B60B          DBL   :BB6
351 0C02 D8D2D9A0      ASC   'XRY '
352 0C06 AE0B          DBL   :BAE
353 0C08 C3CDD0D9      ASC   'CMPY'
354 0C0C BE0B          DBL   :BBE
355 0C0E C9CECDD9      ASC   'INMY'
356 0C12 340B          DBL   :B34
357 0C14 C4C3CDD9      ASC   'DCMY'
358 0C18 350B          DBL   :B35
359
360 *
361 * CLASS C - 2 BYTES FIXED, DISPLACEMENT, IMM. OP.
362
362 0C1A D3C9D8A0      ASC   'SIX '   , STORE IMMEDIATE THROUGH X
363 0C1E 360C          DBL   :C36
364 0C20 D3C9D9A0      ASC   'SIY '   STORE IMMEDIATE THROUGH Y
365 0C24 360C          DBL   :C36
366
367 * CLASS D - 1 BYTE FIXED
368
369 0C26 D8C1C6A0      ASC   'XAF '   EXCHANGE A,F WITH A',F'
370 0C2A 088D          DBL   :8D88
371 0C2C C5D8D8A0      ASC   'EXX '   EXCHANGE B-L WITH B'-L'
372 0C30 D98D          DBL   :8DD9
373
374 * CLASS E - 1 BYTE - SKELETON + R.P.
375
376 0C32 C4C1C3A0      ASC   'DAC '   DOUBLE ADD WITH CARRY TO H&L
377 0C36 0A0E          DBL   :E0A
378 0C38 C4D3C3A0      ASC   'DSC '   DOUBLE SUBTRACT WITH CARRY F
379 0C3C 020E          DBL   :E02
380
381 * CLASS F - 2 BYTES FIXED + 16 BIT OPERAND
382
383 * NULL SET. FED BY 8080 CLASS 5
384
385
386 * CLASS 10 - 2 BYTES FIXED + 16 BIT OPERAND
387
388 0C3E CCC2C3C4      ASC   'LBCD'   LOAD B-C DIRECT
389 0C42 4B10          DBL   :104B
390 0C44 CCC4C5C4      ASC   'LDED'   LOAD D-E DIRECT
391 0C48 5B10          DBL   :105B
392 0C4A CCD3D0C4      ASC   'LSPD'   LOAD STACK POINTER DIRECT
393 0C4E 7B10          DBL   :107B

```



|                   |     |                                              |                            |
|-------------------|-----|----------------------------------------------|----------------------------|
| 394 0C50 D3C2C3C4 | ASC | 'SBCD'                                       | STORE B-C DIRECT           |
| 395 0C54 4310     | DBL | :1043                                        |                            |
| 396 0C56 D3C4C5C4 | ASC | 'SDED'                                       | STORE D-E DIRECT           |
| 397 0C5A 5310     | DBL | :1053                                        |                            |
| 398 0C5C D3D3D0C4 | ASC | 'SSPD'                                       | STORE STACK POINTER DIRECT |
| 399 0C60 7310     | DBL | :1073                                        |                            |
| 400 0C62 CCD8C4A0 | ASC | 'LXD'                                        | LOAD X DIRECT              |
| 401 0C66 2A10     | DBL | :102A                                        |                            |
| 402 0C68 D3D8C4A0 | ASC | 'SXD'                                        | STORE X DIRECT             |
| 403 0C6C 2210     | DBL | :1022                                        |                            |
| 404 0C6E CCD9C4A0 | ASC | 'LYD'                                        | LOAD Y DIRECT              |
| 405 0C72 2A10     | DBL | :102A                                        |                            |
| 406 0C74 D3D9C4A0 | ASC | 'SYD'                                        | STORE Y DIRECT             |
| 407 0C78 2210     | DBL | :1022                                        |                            |
| 408               | *   |                                              |                            |
| 409               | *   | CLASS 11 - 2 BYTES FIXED WITH X OR Y OPERAND |                            |
| 410               | *   |                                              |                            |
| 411               | *   | NULL SET FED BY 8080 CLASS 6                 |                            |
| 412               | *   |                                              |                            |
| 413               | *   |                                              |                            |
| 414               | *   | CLASS 12 - 2 BYTES FIXED, ED PREFIX          |                            |
| 415               | *   |                                              |                            |
| 416 0C7A D4C1C9A0 | ASC | 'TAI'                                        |                            |
| 417 0C7E 5792     | DBL | :9257                                        |                            |
| 418 0C80 D4C1D2A0 | ASC | 'TAR'                                        |                            |
| 419 0C84 5F92     | DBL | :925F                                        |                            |
| 420 0C86 D4C9C1A0 | ASC | 'TIA'                                        |                            |
| 421 0C8A 4792     | DBL | :9247                                        |                            |
| 422 0C8C D4D2C1A0 | ASC | 'TRA'                                        |                            |
| 423 0C90 4F92     | DBL | :924F                                        |                            |
| 424 0C92 D2C5D4C9 | ASC | 'RETI'                                       |                            |
| 425 0C96 4D92     | DBL | :924D                                        |                            |
| 426 0C98 D2C5D4CE | ASC | 'RETN'                                       |                            |
| 427 0C9C 4592     | DBL | :9245                                        |                            |
| 428 0C9E C9CEABA0 | ASC | 'IN+'                                        |                            |
| 429 0CA2 A292     | DBL | :92A2                                        |                            |
| 430 0CA4 C9CEABD2 | ASC | 'IN+R'                                       |                            |
| 431 0CA8 BA92     | DBL | :92BA                                        |                            |
| 432 0CAA C9CEADA0 | ASC | 'IN-'                                        |                            |
| 433 0CAE AA92     | DBL | :92AA                                        |                            |
| 434 0CB0 C9CEADD2 | ASC | 'IN-R'                                       |                            |
| 435 0CB4 BA92     | DBL | :92BA                                        |                            |
| 436 0CB6 CFD5D4AB | ASC | 'OUT+'                                       |                            |
| 437 0CBA A392     | DBL | :92A3                                        |                            |
| 438 0CBC CFD4ABD2 | ASC | 'OUT+R'                                      |                            |
| 439 0CC0 B392     | DBL | :92B3                                        |                            |
| 440 0CC2 CFD5D4AD | ASC | 'OUT-'                                       |                            |
| 441 0CC6 AB92     | DBL | :92AB                                        |                            |
| 442 0CC8 CFD4ADD2 | ASC | 'OUT-R'                                      |                            |
| 443 0CCC BB92     | DBL | :92BB                                        |                            |
| 444 0CCE C9CECIA0 | ASC | 'INA'                                        |                            |
| 445 0CD2 7892     | DBL | :9278                                        |                            |
| 446 0CD4 C9CEC2A0 | ASC | 'INB'                                        |                            |
| 447 0CD8 4092     | DBL | :9240                                        |                            |
| 448 0CDA C9CEC3A0 | ASC | 'INC'                                        |                            |
| 449 0CDE 4892     | DBL | :9248                                        |                            |
| 450 0CE0 C9CEC4A0 | ASC | 'IND'                                        |                            |
| 451 0CE4 5092     | DBL | :9250                                        |                            |

```

452 0CE6 C9CEC5A0      ASC  'INE '
453 0CEA 5892          DBL  :9258
454 0CEC C9CEC8A0      ASC  'INH '
455 0CF0 6092          DBL  :9260
456 0CF2 C9CECCA0      ASC  'INL '
457 0CF6 6892          DBL  :9268
458 0CF8 CFD4C1A0      ASC  'OTA '
459 0CFC 7992          DBL  :9279
460 0CFE CFD4C2A0      ASC  'OTB '
461 0D02 4192          DBL  :9241
462 0D04 CFD4C3A0      ASC  'OTC '
463 0D08 4992          DBL  :9249
464 0D0A CFD4C4A0      ASC  'OTD '
465 0D0E 5192          DBL  :9251
466 0D10 CFD4C5A0      ASC  'OTE '
467 0D14 5992          DBL  :9259
468 0D16 CFD4C8A0      ASC  'OTH '
469 0D1A 6192          DBL  :9261
470 0D1C CFD4CCA0      ASC  'OTL '
471 0D20 6992          DBL  :9269
472 0D22 CDD7ABA0      ASC  'MW+ '
473 0D26 A092          DBL  :92A0
474 0D28 CDD7ABD2      ASC  'MW+R'
475 0D2C B092          DBL  :92B0
476 0D2E CDD7ADA0      ASC  'MW- '
477 0D32 A892          DBL  :92A8
478 0D34 CDD7ADD2      ASC  'MW-R'
479 0D38 B892          DBL  :92B8
480 0D3A C3D0ABA0      ASC  'CP+ '
481 0D3E A192          DBL  :92A1
482 0D40 C3D0ABD2      ASC  'CP+R'
483 0D44 B192          DBL  :92B1
484 0D46 C3D0ADA0      ASC  'CP- '
485 0D4A A992          DBL  :92A9
486 0D4C C3D0ADD2      ASC  'CP-R'
487 0D50 B992          DBL  :92B9
488 0D52 CEC5C7A0      ASC  'NEG '
489 0D56 4492          DBL  :9244
490 0D58 C9CDB0A0      ASC  'IM0 '
491 0D5C 4692          DBL  :9246
492 0D5E C9CDB1A0      ASC  'IM1 '
493 0D62 5692          DBL  :9256
494 0D64 C9CDB2A0      ASC  'IM2 '
495 0D68 5E92          DBL  :925E
496 0D6A D2CCC4A0      ASC  'RLD '
497 0D6E 6F92          DBL  :926F
498 0D70 D2D2C4A0      ASC  'RRD '
499 0D74 6792          DBL  :9267

```

```

*
* CLASS 13 - 2 BYTES FIXED, CB PREFIX
*

```

```

500
501
502
503 0D76 D2D2C3C2      ASC  'RRCB'
504 0D7A 0893          DBL  :9308
505 0D7C D2D2C3C3      ASC  'RRCC'
506 0D80 0993          DBL  :9309
507 0D82 D2D2C3C4      ASC  'RRCD'
508 0D86 0A93          DBL  :930A
509 0D88 D2D2C3C5      ASC  'RRCE'

```

|                   |            |
|-------------------|------------|
| 510 0D8C 0B93     | DBL :930B  |
| 511 0D8E D2D2C3C8 | ASC 'RRCH' |
| 512 0D92 0C93     | DBL :930C  |
| 513 0D94 D2D2C3CC | ASC 'RRCL' |
| 514 0D98 0D93     | DBL :930D  |
| 515 0D9A D2D2C3CD | ASC 'RRCM' |
| 516 0D9E 0E93     | DBL :930E  |
| 517 0DA0 D2D2C3C1 | ASC 'RRCA' |
| 518 0DA4 0F93     | DBL :930F  |
| 519 0DA6 D2C2D2A0 | ASC 'RBR ' |
| 520 0DAA 1B93     | DBL :9318  |
| 521 0DAC D2C3D2A0 | ASC 'RCR ' |
| 522 0DB0 1993     | DBL :9319  |
| 523 0DB2 D2C4D2A0 | ASC 'RDR ' |
| 524 0DB6 1A93     | DBL :931A  |
| 525 0DB8 D2C5D2A0 | ASC 'RER ' |
| 526 0DBC 1B93     | DBL :931B  |
| 527 0DBE D2C8D2A0 | ASC 'RHR ' |
| 528 0DC2 1C93     | DBL :931C  |
| 529 0DC4 D2CCD2A0 | ASC 'RLR ' |
| 530 0DC8 1D93     | DBL :931D  |
| 531 0DCA D2CDD2A0 | ASC 'RMR ' |
| 532 0DCE 1E93     | DBL :931E  |
| 533 0DD0 D2C1D2A4 | ASC 'RAR#' |
| 534 0DD4 1F93     | DBL :931F  |
| 535 0DD6 CCCCC2A0 | ASC 'LLB ' |
| 536 0DDA 2093     | DBL :9320  |
| 537 0DDC CCCCC3A0 | ASC 'LLC ' |
| 538 0DE0 2193     | DBL :9321  |
| 539 0DE2 CCCCC4A0 | ASC 'LLD ' |
| 540 0DE6 2293     | DBL :9322  |
| 541 0DE8 CCCCC5A0 | ASC 'LLE ' |
| 542 0DEC 2393     | DBL :9323  |
| 543 0DEE CCCCC8A0 | ASC 'LLH ' |
| 544 0DF2 2493     | DBL :9324  |
| 545 0DF4 CCCCCCA0 | ASC 'LLL ' |
| 546 0DF8 2593     | DBL :9325  |
| 547 0DFA CCCCCDA0 | ASC 'LLM ' |
| 548 0DFE 2693     | DBL :9326  |
| 549 0E00 CCCCC1A0 | ASC 'LLA ' |
| 550 0E04 2793     | DBL :9327  |
| 551 0E06 D2CCC3C2 | ASC 'RLCB' |
| 552 0E0A 0093     | DBL :9300  |
| 553 0E0C D2CCC3C3 | ASC 'RLCC' |
| 554 0E10 0193     | DBL :9301  |
| 555 0E12 D2CCC3C4 | ASC 'RLCD' |
| 556 0E16 0293     | DBL :9302  |
| 557 0E18 D2CCC3C5 | ASC 'RLCE' |
| 558 0E1C 0393     | DBL :9303  |
| 559 0E1E D2CCC3C8 | ASC 'RLCH' |
| 560 0E22 0493     | DBL :9304  |
| 561 0E24 D2CCC3CC | ASC 'RLCL' |
| 562 0E28 0593     | DBL :9305  |
| 563 0E2A D2CCC3CD | ASC 'RLCM' |
| 564 0E2E 0693     | DBL :9306  |
| 565 0E30 D2CCC3C1 | ASC 'RLCA' |
| 566 0E34 0793     | DBL :9307  |
| 567 0E36 D2C2CCA0 | ASC 'RBL ' |

Z80 ROTATE RIGHT

|                   |     |         |
|-------------------|-----|---------|
| 568 0E3A 1093     | DBL | :9310   |
| 569 0E3C D2C3CCA0 | ASC | 'RCL '  |
| 570 0E40 1193     | DBL | :9311   |
| 571 0E42 D2C4CCA0 | ASC | 'RDL '  |
| 572 0E46 1293     | DBL | :9312   |
| 573 0E48 D2C5CCA0 | ASC | 'REL '  |
| 574 0E4C 1393     | DBL | :9313   |
| 575 0E4E D2C8CCA0 | ASC | 'RHL '  |
| 576 0E52 1493     | DBL | :9314   |
| 577 0E54 D2CCCCA0 | ASC | 'RLL '  |
| 578 0E58 1593     | DBL | :9315   |
| 579 0E5A D2CDCCA0 | ASC | 'RML '  |
| 580 0E5E 1693     | DBL | :9316   |
| 581 0E60 D2C1CCA4 | ASC | 'RAL\$' |
| 582 0E64 1793     | DBL | :9317   |
| 583 0E66 C1D2C2A0 | ASC | 'ARB '  |
| 584 0E6A 2893     | DBL | :9328   |
| 585 0E6C C1D2C3A0 | ASC | 'ARC '  |
| 586 0E70 2993     | DBL | :9329   |
| 587 0E72 C1D2C4A0 | ASC | 'ARD '  |
| 588 0E76 2A93     | DBL | :932A   |
| 589 0E78 C1D2C5A0 | ASC | 'ARE '  |
| 590 0E7C 2B93     | DBL | :932B   |
| 591 0E7E C1D2C8A0 | ASC | 'ARH '  |
| 592 0E82 2C93     | DBL | :932C   |
| 593 0E84 C1D2CCA0 | ASC | 'ARL '  |
| 594 0E88 2D93     | DBL | :932D   |
| 595 0E8A C1D2CDA0 | ASC | 'ARM '  |
| 596 0E8E 2E93     | DBL | :932E   |
| 597 0E90 C1D2C1A0 | ASC | 'ARA '  |
| 598 0E94 2F93     | DBL | :932F   |
| 599 0E96 CCD2C2A0 | ASC | 'LRB '  |
| 600 0E9A 3893     | DBL | :9338   |
| 601 0E9C CCD2C3A0 | ASC | 'LRC '  |
| 602 0EA0 3993     | DBL | :9339   |
| 603 0EA2 CCD2C4A0 | ASC | 'LRD '  |
| 604 0EA6 3A93     | DBL | :933A   |
| 605 0EA8 CCD2C5A0 | ASC | 'LRE '  |
| 606 0EAC 3B93     | DBL | :933B   |
| 607 0EAE CCD2C8A0 | ASC | 'LRH '  |
| 608 0EB2 3C93     | DBL | :933C   |
| 609 0EB4 CCD2CCA0 | ASC | 'LRL '  |
| 610 0EB8 3D93     | DBL | :933D   |
| 611 0EBA CCD2CDA0 | ASC | 'LRM '  |
| 612 0EBE 3E93     | DBL | :933E   |
| 613 0EC0 CCD2C1A0 | ASC | 'LRA '  |
| 614 0EC4 3F93     | DBL | :933F   |
| 615               |     |         |
| 616               |     |         |
| 617               |     |         |
| 618 0EC6 C2D2C1A0 | ASC | 'BRA '  |
| 619 0ECA 1814     | DBL | :1418   |
| 620 0ECC C2C3A0A0 | ASC | 'BC '   |
| 621 0ED0 3814     | DBL | :1438   |
| 622 0ED2 C2CCD3A0 | ASC | 'BLS '  |
| 623 0ED6 3814     | DBL | :1438   |
| 624 0ED8 C2CEC3A0 | ASC | 'BNC '  |
| 625 0EDC 3014     | DBL | :1430   |

Z80 ROTATE LEFT

\*  
\* CLASS 14 - RELATIVE BRANCHES  
\*

|                   |                                        |        |
|-------------------|----------------------------------------|--------|
| 626 0EDE C2C7C5A0 | ASC                                    | 'BGE ' |
| 627 0EE2 3014     | DBL                                    | :1430  |
| 628 0EE4 C2DAA0A0 | ASC                                    | 'BZ '  |
| 629 0EE8 2814     | DBL                                    | :1428  |
| 630 0EEA C2C5D1A0 | ASC                                    | 'BEQ ' |
| 631 0EEE 2814     | DBL                                    | :1428  |
| 632 0EF0 C2CEDAA0 | ASC                                    | 'BNZ ' |
| 633 0EF4 2014     | DBL                                    | :1420  |
| 634 0EF6 C2CEC5A0 | ASC                                    | 'BNE ' |
| 635 0EFA 2014     | DBL                                    | :1420  |
| 636 0EFC C4C2CEDA | ASC                                    | 'DBNZ' |
| 637 0F00 1014     | DBL                                    | :1410  |
| 638               |                                        |        |
| 639               | * CLASS 15 - REGISTER BIT MANIPULATION |        |
| 640               | * PREFIX CB                            |        |
| 641               | *                                      |        |
| 642 0F02 C2D4C2A0 | ASC                                    | 'BTB ' |
| 643 0F06 4015     | DBL                                    | :1540  |
| 644 0F08 C2D4C3A0 | ASC                                    | 'BTC ' |
| 645 0F0C 4115     | DBL                                    | :1541  |
| 646 0F0E C2D4C4A0 | ASC                                    | 'BTD ' |
| 647 0F12 4215     | DBL                                    | :1542  |
| 648 0F14 C2D4C5A0 | ASC                                    | 'BTE ' |
| 649 0F18 4315     | DBL                                    | :1543  |
| 650 0F1A C2D4C8A0 | ASC                                    | 'BTH ' |
| 651 0F1E 4415     | DBL                                    | :1544  |
| 652 0F20 C2D4CCA0 | ASC                                    | 'BTL ' |
| 653 0F24 4515     | DBL                                    | :1545  |
| 654 0F26 C2D4CDA0 | ASC                                    | 'BTM ' |
| 655 0F2A 4615     | DBL                                    | :1546  |
| 656 0F2C C2D4C1A0 | ASC                                    | 'BTA ' |
| 657 0F30 4715     | DBL                                    | :1547  |
| 658 0F32 C2D3C2A0 | ASC                                    | 'BSB ' |
| 659 0F36 C015     | DBL                                    | :15C0  |
| 660 0F38 C2D3C3A0 | ASC                                    | 'BSC ' |
| 661 0F3C C115     | DBL                                    | :15C1  |
| 662 0F3E C2D3C4A0 | ASC                                    | 'BSD ' |
| 663 0F42 C215     | DBL                                    | :15C2  |
| 664 0F44 C2D3C5A0 | ASC                                    | 'BSE ' |
| 665 0F48 C315     | DBL                                    | :15C3  |
| 666 0F4A C2D3C8A0 | ASC                                    | 'BSH ' |
| 667 0F4E C315     | DBL                                    | :15C3  |
| 668 0F50 C2D3CCA0 | ASC                                    | 'BSL ' |
| 669 0F54 C515     | DBL                                    | :15C5  |
| 670 0F56 C2D3CDA0 | ASC                                    | 'BSM ' |
| 671 0F5A C615     | DBL                                    | :15C6  |
| 672 0F5C C2D3C1A0 | ASC                                    | 'BSA ' |
| 673 0F60 C715     | DBL                                    | :15C7  |
| 674 0F62 C2C3C2A0 | ASC                                    | 'BCB ' |
| 675 0F66 8015     | DBL                                    | :1580  |
| 676 0F68 C2C3C3A0 | ASC                                    | 'BCC ' |
| 677 0F6C 8115     | DBL                                    | :1581  |
| 678 0F6E C2C3C4A0 | ASC                                    | 'BCD ' |
| 679 0F72 8215     | DBL                                    | :1582  |
| 680 0F74 C2C3C5A0 | ASC                                    | 'BCE ' |
| 681 0F78 8315     | DBL                                    | :1583  |
| 682 0F7A C2C3C8A0 | ASC                                    | 'BCH ' |
| 683 0F7E 8415     | DBL                                    | :1584  |



```

684 0F80 C2C3CCA0      ASC  'BCL '
685 0F84 8515          DBL  :1585
686 0F86 C2C3CDA0      ASC  'BCM '
687 0F8A 8615          DBL  :1586
688 0F8C C2C3CIA0      ASC  'BCA '
689 0F90 8715          DBL  :1587
690
691                      *
692                      * CLASS 16 - SHIFT/ROTATE MEMORY THROUGH X OR Y
693                      *
693 0F92 D2CCC3D8      ASC  'RLCX'
694 0F96 0616          DBL  :1606
695 0F98 D2CCC3D9      ASC  'RLCY'
696 0F9C 0616          DBL  :1606
697 0F9E D2CDDCCD8     ASC  'RMLX'
698 0FA2 1616          DBL  :1616
699 0FA4 D2CDDCCD9     ASC  'RMLY'
700 0FA8 1616          DBL  :1616
701 0FAA D2D2C3D8      ASC  'RRCX'
702 0FAE 0E16          DBL  :160E
703 0FB0 D2D2C3D9      ASC  'RRCY'
704 0FB4 0E16          DBL  :160E
705 0FB6 D2CDD2D8      ASC  'RMRX'
706 0FBA 1E16          DBL  :161E
707 0FBC D2CDD2D9      ASC  'RMRY'
708 0FB8 1E16          DBL  :161E
709 0FC2 CCCCCDD8      ASC  'LLMX'
710 0FC6 2616          DBL  :1626
711 0FC8 CCCCCDD9      ASC  'LLMY'
712 0FCC 2616          DBL  :1626
713 0FCE C1D2CDD8      ASC  'ARMX'
714 0FD2 2E16          DBL  :162E
715 0FD4 C1D2CDD9      ASC  'ARMY'
716 0FD8 2E16          DBL  :162E
717 0FDA CDD2CDD8      ASC  'LRMX'
718 0FDE 3E16          DBL  :163E
719 0FE0 CDD2CDD9      ASC  'LRMY'
720 0FE4 3E16          DBL  :163E
721
722                      *
723                      * CLASS 17 BIT MANIPULATION THROUGH X OR Y
724                      *
724 0FE6 C2D4CDD8      ASC  'BTMX'
725 0FEA 4617          DBL  :1746
726 0FEC C2D4CDD9      ASC  'BTMY'
727 0FF0 4617          DBL  :1746
728 0FF2 C2D3CDD8      ASC  'BSMX'
729 0FF6 C617          DBL  :17C6
730 0FF8 C2D3CDD9      ASC  'BSMY'
731 0FFC C617          DBL  :17C6
732 0FFE C2C3CDD8      ASC  'BCMx'
733 1002 8617          DBL  :1786
734 1004 C2C3CDD9      ASC  'BCMY'
735 1008 8617          DBL  :1786
736
737                      *
738                      * CLASS 18 - MISC. OPERATIONS INVOLVING X
739                      * PREFIX DD
740                      *
740 100A D8ABC2C3      ASC  'X+BC'
741 100E 0998          DBL  :9809

```

```

742 1010 D8ABC4C5      ASC  'X+DE'
743 1014 1998          DBL  :9819
744 1016 D8ABD8A0      ASC  'X+X '
745 101A 2998          DBL  :9829
746 101C D8ABD3D0      ASC  'X+5P'
747 1020 3998          DBL  :9839
748 1022 D0C3D8A0      ASC  'PCX '
749 1026 E998          DBL  :98E9
750 1028 D8D4D8A0      ASC  'XTX '
751 102C E398          DBL  :98E3
752 102E D3D0D8A0      ASC  'SPX '
753 1032 F998          DBL  :98F9
754
755 * CLASS 19 - MISC. OPERATIONS INVOLVING Y
756 * PREFIX FD
757 *
758 1034 D9ABC2C3      ASC  'Y+BC'
759 1038 0999          DBL  :9909
760 103A D9ABC4C5      ASC  'Y+DE'
761 103E 1999          DBL  :9919
762 1040 D9ABD9A0      ASC  'Y+Y '
763 1044 2999          DBL  :9929
764 1046 D9ABD3D0      ASC  'Y+5P'
765 104A 3999          DBL  :9939
766 104C D0C3D9A0      ASC  'PCY '
767 1050 E999          DBL  :99E9
768 1052 D8D4D9A0      ASC  'XTY '
769 1056 E399          DBL  :99E3
770 1058 D3D0D9A0      ASC  'SPY '
771 105C F999          DBL  :99F9
772
773 *****
774 *
775 * ASSEMBLY MONITOR AND SYMBOL TABLE HANDLERS
776 *
776 105E AF            ASM  ZAR
777 105F 329F13        STA  INFLAG      SET ASM MODE TO MEMORY
778 1062 325F02        STA  NULLS
779 1065 32B813        STA  PAGE
780 1068 328C1F        STA  IL8080
781 106B 3E8D          MVI  A, :8D
782 106D 32C813        STA  TITLE
783 1070 CD7D01        CALL RKB
784 1073 FED4          CPI  'T'        TAPE ASSEMBLY REQUEST
785 1075 C27E10        JNE  ASM2
786 1078 329F13        STA  INFLAG      SET TAPE MODE
787 107B CD7D01        CALL RKB
788 107E 32A213        ASM2 STA  LSTFLG
789 1081 FECC          CPI  'L'
790 1083 CA9210        JEQ  ALIST
791 1086 FECF          CPI  'O'        OBJECT TAPE?
792 1088 CA9110        JEQ  AOBJ
793 108B FEC5          CPI  'E'        ERROR SCAN?
794 108D CA9810        JEQ  ASM2A
795 1090 FEDA          CPI  'Z'
796 1092 C2E002        JNE  ERR
797 1095 328C1F        STA  IL8080
798 1098 AF            ASM2A ZAR
799 1099 32A013        STA  PRFLAG      CLEAR PRINT

```

|                 |       |      |          |                             |
|-----------------|-------|------|----------|-----------------------------|
| 800 109C 32A113 | ASM3  | STA  | OBFLAG   | AND PUNCH FLAGS             |
| 801 109F C3B010 |       | JMP  | ASM3A    |                             |
| 802 10A2 32A013 | ALIST | STA  | PRFLAG   | SET PRINT FLAG              |
| 803 10A5 AF     |       | ZAR  |          |                             |
| 804 10A6 C39C10 |       | JMP  | ASM3     |                             |
| 805 10A9 32A113 | ROBJ  | STA  | OBFLAG   | SET PUNCH FLAG              |
| 806 10AC AF     |       | ZAR  |          |                             |
| 807 10AD 32A013 |       | STA  | PRFLAG   | CLEAR PRINT FLAG            |
| 808 10B0 CD7D01 | ASM3A | CALL | RKB      | GET PERIOD                  |
| 809 10B3 FEAE   |       | CPI  | ' '      |                             |
| 810 10B5 C2E002 |       | JNE  | ERR      |                             |
| 811 10B8 CD8301 |       | CALL | REWIND   |                             |
| 812 10BB 3A9F13 |       | LDA  | INFLAG   | CHECK FOR TAPE INPUT        |
| 813 10BE B7     |       | ORA  | A        |                             |
| 814 10BF C2DC10 |       | JNZ  | ASM4     |                             |
| 815 10C2 2A3B02 |       | LHLD | LNCNT    | MEMORY, CHECK "END"         |
| 816 10C5 7C     |       | MOV  | A,H      |                             |
| 817 10C6 B5     |       | ORA  | L        |                             |
| 818 10C7 CAF114 |       | JZ   | NOPROG   |                             |
| 819 10CA CDAB02 |       | CALL | FBA      |                             |
| 820 10CD CD2106 |       | CALL | FORMAT   |                             |
| 821 10D0 7D     |       | MOV  | A,L      |                             |
| 822 10D1 FE80   |       | CPI  | :80      |                             |
| 823 10D3 C21B12 |       | JNE  | NOEND    |                             |
| 824 10D6 7C     |       | MOV  | A,H      |                             |
| 825 10D7 FE01   |       | CPI  | I        |                             |
| 826 10D9 C21B12 |       | JNE  | NOEND    |                             |
| 827 10DC 2A4702 | ASM4  | LHLD | BFR      | SET MEMORY SOURCE POINTER   |
| 828 10DF 22B913 |       | SHLD | SRCPTR   |                             |
| 829 10E2 3A9F13 |       | LDA  | INFLAG   |                             |
| 830 10E5 B7     |       | ORA  | A        |                             |
| 831 10E6 CAFE10 |       | JZ   | AMEM     |                             |
| 832 10E9 22AF13 |       | SHLD | SYMLOC   | TAPE ASSEMBLY, SET          |
| 833 10EC 22B113 |       | SHLD | SYMPTR   | SYMBOL TABLE POINTERS       |
| 834 10EF 225502 |       | SHLD | NXTCHR   | CLEAR BUFFER                |
| 835 10F2 224502 |       | SHLD | BFA      |                             |
| 836 10F5 210000 |       | LXI  | H,0      |                             |
| 837 10F8 223B02 |       | SHLD | LNCNT    |                             |
| 838 10FB C31711 |       | JMP  | ASM5     |                             |
| 839 10FE 2A5502 | AMEM  | LHLD | NXTCHR   | MEMORY ASSEMBLY, SET        |
| 840 1101 22AF13 |       | SHLD | SYMLOC   | SYMBOL TABLE POINTERS       |
| 841 1104 22B113 |       | SHLD | SYMPTR   |                             |
| 842 1107 3AA113 |       | LDA  | OBFLAG   |                             |
| 843 110A B7     |       | ORA  | A        |                             |
| 844 110B CA1711 |       | JZ   | ASM5     |                             |
| 845 110E 215E11 |       | LXI  | H,RDPMSG |                             |
| 846 1111 CD9901 |       | CALL | PRT      |                             |
| 847 1114 CD7401 |       | CALL | SB       | WAIT TO TURN ON PUNCH       |
| 848 1117 CD0F17 | ASM5  | CALL | PASS1    | EXECUTE PASS 1              |
| 849 111A 3AA013 |       | LDA  | PRFLAG   |                             |
| 850 111D B7     |       | ORA  | A        |                             |
| 851 111E CA2411 |       | JZ   | ASM5A    |                             |
| 852 1121 CD3912 |       | CALL | SRTST    | SORT SYMBOL TABLE           |
| 853 1124 CD8301 | ASM5A | CALL | REWIND   |                             |
| 854 1127 2A4702 |       | LHLD | BFR      |                             |
| 855 112A 22B913 |       | SHLD | SRCPTR   | RESET POINTERS FOR 2ND PASS |
| 856 112D AF     |       | ZAR  |          |                             |
| 857 112E 325F02 |       | STA  | NULLS    | CLEAR TAPE LEADER FLAG      |

```

858 1131 32BC13          STA  LINZ
859 1134 3AA013          LDA  PRFLAG
860 1137 B7              ORA  A
861 1138 CA4111          JZ   ASM6
862 1138 CD9714          CALL EJECT
863 113E CDC914          CALL PGN8
864 1141 CDC619          ASM6 CALL PASS2
865 1144 3AA013          LDA  PRFLAG
866 1147 B7              ORA  A
867 1148 CA5111          JZ   ASM6AA
868 1148 CDB812          CALL PSYMTB   PRINT SORTED TABLE
869 114E CD9714          CALL EJECT   RESTORE PAPER
870 1151 3AA113          ASM6AA LDA  QBFLAG
871 1154 B7              ORA  A
872 1155 CA1901          JZ   MON2
873 1158 CD7401          CALL  SB      WAIT FOR SPACE
874 115B C31901          JMP  MON2      RETURN TO MONITOR
875 115E D2C5C1C4       RDPMSG ASC  'READY PUNCH, SPACE'
876 1170 8D              DATA 8D
877
878
879
880
881
882
883
884
885
886 1171 22AB13          SYMENT SHLD SVAL      SAVE SYMBOL VALUE
887 1174 2A4902          LHL D  EOB      CHECK FOR BUFFER OVERFLOW
888 1177 44              MOV  B,H
889 1178 4D              MOV  C,L
890 1179 2AB113          LHL D  SYMPTR
891 117C 110800          LXI  D,8
892 117F 19              DAD  D
893 1180 CD3304          CALL NEGH L
894 1183 09              DAD  B
895 1184 00              NOP
896 1185 00              NOP
897 1186 D2FF11          JNC  STO      SYMBOL TABLE OVERFLOW
898 1189 2AB113          LHL D  SYMPTR
899 118C EB              XCHG
900 118D 0606          MVI  B,6
901 118F 21D007          LXI  H,LABEL
902 1192 7E              MOV  A,M      MOVE SYMBOL TO TABLE
903 1193 12              STAX D
904 1194 23              INX  H
905 1195 13              INX  D
906 1196 05              DCR  B
907 1197 C29211          JNZ  SE2
908 119A 3AAB13          LDA  SVAL      MOVE VALUE TO TABLE
909 119D 12              STAX D
910 119E 13              INX  D
911 119F 3AAC13          LDA  SVAL+1
912 11A2 12              STAX D
913 11A3 13              INX  D
914 11A4 EB              XCHG
915 11A5 22B113          SHLD SYMPTR  UPDATE SYMBOL POINTER

```

```

916 11A8 2AADI3      LHL D NSYMS      INCREMENT SYMBOL COUNT
917 11AB 23          INX H
918 11AC 22ADI3      SHLD NSYMS
919 11AF C9          RET              RETURN
920
921 *
922 * SEARCH SYMBOL TABLE FOR SYMBOL WHOSE
923 * ADDRESS IS IN H&L ON ENTRY. VALUE OF
924 * SYMBOL RETURNS IN H AND L, CARRY
925 * OFF. IF SYMBOL NOT FOUND CARRY RETURNS ON.
926 11B0 22ABI3      SRCSTB SHLD SVAL      SAVE ADDRESS
927 11B3 EB          XCHG
928 11B4 22A3I3      SHLD DSAVE
929 11B7 2AADI3      LHL D NSYMS      CHECK FOR NO SYMBOLS
930 11BA 7C          MOV A,H
931 11BB B5          ORA L
932 11BC CAF1I1      JZ STMT          SYMBOL TABLE EMPTY
933 11BF 22B3I3      SHLD SYMCNT      SAVE COUNT
934 11C2 2AAF13      LHL D SYMLOC
935 11C5 EB          XCHG
936 11C6 2AABI3      SRST1  LHL D SVAL
937 11C9 010600      LXI B,6
938 11CC EB          XCHG
939 11CD 22A5I3      SHLD STLOC
940 11D0 EB          XCHG
941 11D1 1A          SRST2  LDAX D
942 11D2 96          SUB M
943 11D3 B0          ORA B
944 11D4 47          MOV B,A
945 11D5 23          INX H
946 11D6 13          INX D
947 11D7 00          DCR C
948 11D8 C2D1I1      JNZ SRST2
949 11DB 78          MOV A,B
950 11DC E67F        ANI :7F
951 11DE CAF6I1      JZ FSYMBL
952 11E1 13          INX D
953 11E2 13          INX D
954 11E3 2AB3I3      LHL D SYMCNT
955 11E6 2B          DCX H
956 11E7 22B3I3      SHLD SYMCNT
957 11EA 7C          MOV A,H
958 11EB B5          ORA L
959 11EC C2C6I1      JNZ SRST1
960 11EF 37          STMT  STC          CARRY ON FOR NO HIT
961 11F0 EB          RSTRD  XCHG
962 11F1 2AA3I3      LHL D DSAVE
963 11F4 EB          XCHG
964 11F5 C9          RET
965 11F6 1A          FSYMBL LDAX D
966 11F7 6F          MOV L,A
967 11F8 13          INX D
968 11F9 1A          LDAX D
969 11FA 67          MOV H,A
970 11FB B7          RSC          CLEAR CARRY FOR FOUND SYMBOL
971 11FC C3F0I1      JMP RSTRD
972
973 11FF 2105I2      *  STO  LXI H,STOM

```



```

974 1202 C31E12      JMP    NE2
975 1205 D3D9CDC2    STOM   ASC   'SYMBOL TABLE OVERFLOW'
976 121A 8D          DATA  :8D
977                  *
978 121B 212412      NOEND   LXI   H,NEMSG
979 121E CD9901      NE2     CALL  PRT
980 1221 C31901      JMP     MON2
981 1224 CCC1D3D4    NEMSG   ASC   'LAST LINE NOT "END" '
982 1238 8D          DATA  :8D
983                  *
984                  *
985                  * SORT SYMBOL TABLE INTO ASCENDING ORDER
986                  *
987 1239 2A0D13      SRTST   LHL   NSYMS
988 123C 2B          DCX     H
989 123D 7C          MOV     A,H
990 123E B7          ORA     A
991 123F F8          RM
992 1240 B5          ORA     L      CHECK FOR 1 OR 0
993 1241 C8          RZ          NO SORT IF ONLY ONE SYMBOL
994 1242 22B713      SRTST0  SHLD  NCMP5
995 1245 22B313      SHLD  SYMCNT
996 1248 2A0F13      LHL   SYMLOC
997 124B 22A913      SHLD  SPTR2
998 124E 210000      LXI     H,0
999 1251 22B513      SHLD  NSWAP5   CLEAR SWAP COUNTER
000 1254 2A0913      SRTST1  LHL   SPTR2
001 1257 EB          XCHG
002 1258 210800      LXI     H,8    LOW ADDRESS TO D&E
003 125B 19          DAD     D      INCREMENT TO H
004 125C 22A913      SHLD  SPTR2   FORM HIGH ADDRESS IN H&L
005 125F EB          XCHG          UPDATE HIGH POINTER
006 1260 22A713      SHLD  SPTR    UPDATE LOW POINTER
007 1263 0606        MVI     B,6    SYMBOL POINTER
008 1265 7E          MOV     A,M
009 1266 E67F        ANI     :7F
010 1268 F5          PUSH    PSW
011 1269 1A          LDAX   D
012 126A E67F        ANI     :7F
013 126C 4F          MOV     C,A
014 126D F1          POP     PSW
015 126E B9          CMP     C
016 126F DA9C12      JAL     SRTST3
017 1272 CA7812      JEQ     SRT2A
018 1275 C38112      JMP     SRTSWP
019 1278 13          INX     D
020 1279 23          INX     H
021 127A 05          DCR     B
022 127B C26512      JNZ     SRTST2
023 127E C39C12      JMP     SRTST3
024 1281 2A0913      SRTSWP  LHL   SPTR2   SWAP ENTRIES
025 1284 EB          XCHG
026 1285 2A0713      LHL   SPTR
027 1288 0608        MVI     B,8
028 128A 1A          LDAX   D
029 128B 4E          MOV     C,M
030 128C 77          MOV     M,A
031 128D 79          MOV     A,C

```

|     |      |        |         |              |                             |
|-----|------|--------|---------|--------------|-----------------------------|
| 032 | 128E | 12     | STAX    | D            |                             |
| 033 | 128F | 23     | INX     | H            |                             |
| 034 | 1290 | 13     | INX     | D            |                             |
| 035 | 1291 | 05     | DCR     | B            |                             |
| 036 | 1292 | C28A12 | JNZ     | SWAP2        |                             |
| 037 | 1295 | 2AB513 | LHLD    | NSWAPS       | INCREMENT SWAP COUNT        |
| 038 | 1298 | 23     | INX     | H            |                             |
| 039 | 1299 | 22B513 | SHLD    | NSWAPS       |                             |
| 040 | 129C | 2AB313 | LHLD    | SYMCNT       | DECREMENT LOOP COUNT        |
| 041 | 129F | 2B     | DCX     | H            |                             |
| 042 | 12A0 | 22B313 | SHLD    | SYMCNT       |                             |
| 043 | 12A3 | 7C     | MOV     | A,H          |                             |
| 044 | 12A4 | B5     | ORA     | L            |                             |
| 045 | 12A5 | C25412 | JNZ     | SRTST1       | DO IT AGAIN, SAM            |
| 046 | 12A8 | 2AB513 | LHLD    | NSWAPS       | END OF PASS, CHECK SWAPS    |
| 047 | 12AB | 7C     | MOV     | A,H          |                             |
| 048 | 12AC | B5     | ORA     | L            |                             |
| 049 | 12AD | C8     | RZ      |              | SORT COMPLETE IF NO SWAPS   |
| 050 | 12AE | 2AB713 | LHLD    | NCMP5        | DECREMENT MASTER LOOP COUNT |
| 051 | 12B1 | 2B     | DCX     | H            |                             |
| 052 | 12B2 | 7C     | MOV     | A,H          |                             |
| 053 | 12B3 | B5     | ORA     | L            | CHECK FOR ZERO              |
| 054 | 12B4 | C8     | RZ      |              | RETURN IF NO COMPARES LEFT  |
| 055 | 12B5 | C34212 | JMP     | SRTST0       | OTHERWISE DO IT AGAIN       |
| 056 |      |        |         |              |                             |
| 057 |      |        | *       |              |                             |
| 058 |      |        | * PRINT | SYMBOL TABLE |                             |
| 059 | 12B8 | 2AAD13 | PSYMTB  | LHLD         | NSYMS                       |
| 060 | 12BB | 7C     |         | MOV          | A,H                         |
| 061 | 12BC | B5     |         | ORA          | L                           |
| 062 | 12BD | C8     |         | RZ           |                             |
| 063 | 12BE | 22B313 |         | SHLD         | SYMCNT                      |
| 064 | 12C1 | 2AAF13 |         | LHLD         | SYMLOC                      |
| 065 | 12C4 | EB     |         | XCHG         |                             |
| 066 | 12C5 | 214B13 |         | LXI          | H,BLKLIN                    |
| 067 | 12C8 | CD6C14 |         | CALL         | LPNT                        |
| 068 | 12CB | 216913 |         | LXI          | H,ASTLIN                    |
| 069 | 12CE | CD6C14 |         | CALL         | LPNT                        |
| 070 | 12D1 | 214D13 |         | LXI          | H,STBMSG                    |
| 071 | 12D4 | CD6C14 |         | CALL         | LPNT                        |
| 072 | 12D7 | 216913 |         | LXI          | H,ASTLIN                    |
| 073 | 12DA | CD6C14 |         | CALL         | LPNT                        |
| 074 | 12DD | 214B13 |         | LXI          | H,BLKLIN                    |
| 075 | 12E0 | CD6C14 |         | CALL         | LPNT                        |
| 076 | 12E3 | CD6801 | PSY1    | CALL         | DTCTCC                      |
| 077 | 12E6 | DAA422 |         | JC           | RTMON                       |
| 078 | 12E9 | 216202 |         | LXI          | H,KB                        |
| 079 | 12EC | 063C   |         | MVI          | B,60                        |
| 080 | 12EE | 3EA0   |         | MVI          | A,                          |
| 081 | 12F0 | 77     | PSY2    | MOV          | M,A                         |
| 082 | 12F1 | 23     |         | INX          | H                           |
| 083 | 12F2 | 05     |         | DCR          | B                           |
| 084 | 12F3 | C2F012 |         | JNZ          | PSY2                        |
| 085 | 12F6 | 216202 |         | LXI          | H,KB                        |
| 086 | 12F9 | 0E04   |         | MVI          | C,4                         |
| 087 | 12FB | 0606   | PSY2A   | MVI          | B,6                         |
| 088 | 12FD | 1A     | PSY3    | LDAX         | D                           |
| 089 | 12FE | 77     |         | MOV          | M,A                         |

```

090 12FF 13          INX  D
091 1300 23          INX  H
092 1301 05          DCR  B
093 1302 C2FD12      JNZ  PSY3
094 1305 3EA0        MVI  A, ' '      SET BLANK BETWEEN SYMBOL & V
095 1307 77          MOV  M,A
096 1308 23          INX  H
097 1309 1A          LDAX D          GET HIGH VALUE BYTE
098 130A 13          INX  D
099 130B F5          PUSH PSW
100 130C 1A          LDAX D
101 130D 13          INX  D
102 130E CD8513      CALL CNV2HX
103 1311 77          MOV  M,A
104 1312 23          INX  H
105 1313 70          MOV  M,B
106 1314 23          INX  H
107 1315 F1          POP  PSW
108 1316 CD8513      CALL CNV2HX
109 1319 77          MOV  M,A
110 131A 23          INX  H
111 131B 70          MOV  M,B
112 131C 23          INX  H
113 131D 3EA0        MVI  A, ' '
114 131F 77          MOV  M,A
115 1320 23          INX  H
116 1321 77          MOV  M,A      SET SPACE BETWEEN SYMBOLS
117 1322 23          INX  H
118 1323 77          MOV  M,A
119 1324 23          INX  H
120 1325 E5          PUSH H
121 1326 2AB313      LHLD SYMCNT
122 1329 2B          DCX  H
123 132A 22B313      SHLD SYMCNT
124 132D 7C          MOV  A,H      CHECK FOR LAST SYMBOL
125 132E 85          ORA  L
126 132F E1          POP  H
127 1330 CA3D13      JZ   PSYEND
128 1333 0D          DCR  C          LAST ON LINE?
129 1334 C2FB12      JNZ  PSY2A
130 1337 CD4113      CALL PSYLN      PRINT LINE
131 133A C3E312      JMP  PSY1
132 133D CD4113      PSYEND CALL PSYLN
133 1340 C9          RET
134 1341 3E8D        PSYLN MVI  A, :8D      SET TERMINATOR
135 1343 77          MOV  M,A
136 1344 216202      LXI  H,KB
137 1347 CD6C14      CALL LPNT
138 134A C9          RET
139 134B A0          BLKLIN DATA :A0
140 134C 8D          DATA :8D
141 134D AAA0D3A0     STBMSG ASC  '* S Y M B O L   T A B L E *'
142 1368 8D          DATA :8D
143 1369 AAAAAAAA     ASTLIN ASC  '*****'
144 1384 8D          DATA :8D
145
146 * CONVERT BINARY A TO 2 HEX CHARACTERS
147 *

```

|                   |                                  |      |         |                            |
|-------------------|----------------------------------|------|---------|----------------------------|
| 148 1385 F5       | CNV2HX                           | PUSH | PSW     | SAVE CHARACTER             |
| 149 1386 CD9313   |                                  | CALL | ZHEX    |                            |
| 150 1389 47       |                                  | MOV  | B,A     |                            |
| 151 138A F1       |                                  | PDP  | PSW     | RETRIEVE CHARACTER         |
| 152 138B 1F       |                                  | RAR  |         |                            |
| 153 138C 1F       |                                  | RAR  |         |                            |
| 154 138D 1F       |                                  | RAR  |         |                            |
| 155 138E 1F       |                                  | RAR  |         |                            |
| 156 138F CD9313   |                                  | CALL | ZHEX    |                            |
| 157 1392 C9       |                                  | RET  |         |                            |
| 158 1393 E60F     | ZHEX                             | ANI  | :F      |                            |
| 159 1395 FE0A     |                                  | CPI  | 10      |                            |
| 160 1397 DA9C13   |                                  | JAL  | ZH2     |                            |
| 161 139A C607     |                                  | ADI  | 7       |                            |
| 162 139C C6B0     | ZH2                              | ADI  | :B0     |                            |
| 163 139E C9       |                                  | RET  |         |                            |
| 164               | *                                |      |         |                            |
| 165 139F 00       | INFLAG                           | ***  |         |                            |
| 166 13A0 00       | PRFLAG                           | ***  |         |                            |
| 167 13A1 00       | OBFLAG                           | ***  |         |                            |
| 168 13A2 00       | LSTFLG                           | ***  |         |                            |
| 169 13A3 0000     | DSAVE                            | DBL  | 0       |                            |
| 170 13A5 0000     | STLOC                            | DBL  | 0       |                            |
| 171 13A7 0000     | SPTR                             | DBL  | 0       |                            |
| 172 13A9 0000     | SPTR2                            | DBL  | 0       |                            |
| 173 13AB 0000     | SVAR                             | DBL  | 0       |                            |
| 174 13AD 0000     | NSYMS                            | DBL  | 0       |                            |
| 175 13AF 0000     | SYMLOC                           | DBL  | 0       |                            |
| 176 13B1 0000     | SYMPTR                           | DBL  | 0       |                            |
| 177 13B3 0000     | SYMCNT                           | DBL  | 0       |                            |
| 178 13B5 0000     | NSWAPS                           | DBL  | 0       |                            |
| 179 13B7 0000     | NCMPS                            | DBL  | 0       |                            |
| 180 13B9 0000     | SRCPTR                           | DBL  | 0       |                            |
| 181 13BB 00       | PAGE                             | ***  |         |                            |
| 182 13BC 00       | LINZ                             | ***  |         |                            |
| 183 13BD D0C1C7C5 | PGNO                             | ASC  | 'PAGE ' |                            |
| 184 13C2 00       | PD1                              | ***  |         |                            |
| 185 13C3 00       | PD2                              | ***  |         |                            |
| 186 13C4 A0A0A0A0 | ASC                              |      |         |                            |
| 187 13C8          | TITLE                            | RES  | 38      |                            |
| 188 13EE 8D       |                                  | DATA | :8D     |                            |
| 189 13EF          | TBUF                             | RES  | 72      |                            |
| 190               | *                                |      |         |                            |
| 191               | * FETCH SOURCE RECORD AND FORMAT |      |         |                            |
| 192               | *                                |      |         |                            |
| 193 1437 3A9F13   | FSRCD                            | LDA  | INFLAG  |                            |
| 194 143A B7       |                                  | ORA  | A       |                            |
| 195 143B CA5814   |                                  | JZ   | FMRCD   | RECORD IN MEMORY           |
| 196 143E CDC505   |                                  | CALL | RTR     | GET RECORD FROM TAPE       |
| 197 1441 216202   |                                  | LXI  | H,KB    |                            |
| 198 1444 11EF13   |                                  | LXI  | D,TBUF  | MOVE RECORD TO WORK BUFFER |
| 199 1447 0648     |                                  | MVI  | B,72    |                            |
| 200 1449 7E       | FSRCD2                           | MOV  | A,M     |                            |
| 201 144A 12       |                                  | STAX | D       |                            |
| 202 144B 23       |                                  | INX  | H       |                            |
| 203 144C 13       |                                  | INX  | D       |                            |
| 204 144D 05       |                                  | DCR  | B       |                            |
| 205 144E C24914   |                                  | JNZ  | FSRCD2  |                            |

```

206 1451 21EF13          LXI  H, TBUF
207 1454 CD2106          CALL FORMAT      FORMAT THE RECORD
208 1457 C9              RET
209 1458 2AB913          FMRCO  LHL  SRCPTR
210 145B CD2106          CALL FORMAT
211 145E 2AB913          LHL  SRCPTR      UPDATE SOURCE POINTER
212 1461 7E              FMRCO2 MOV  A, M
213 1462 23              INX  H
214 1463 FE8D            CPI   :8D
215 1465 C26114          JNE  FMRCO2
216 1468 22B913          SHLD SRCPTR
217 146B C9              RET
218
219
220
221
222
223
224
225
226
227
228 146C 3E8D            LPNT  MVI  A, :8D
229 146E CD7701          CALL  CPNT
230 1471 3E8A            MVI  A, :8A
231 1473 CD7701          CALL  CPNT
232 1476 3EFF            MVI  A, :FF
233 1478 CD7701          CALL  CPNT
234 147B 7E              LP3   MOV  A, M
235 147C 23              INX  H
236 147D FE8D            CPI   :8D
237 147F CA8814          JZ    LP2
238 1482 CD7701          CALL  CPNT
239 1485 C37B14          JMP   LP3
240 1488 3ABC13          LP2   LDA  LINZ
241 148B 3D              DCR   A
242 148C 32BC13          STA  LINZ
243 148F C0              RNZ
244 1490 CD9714          CALL  EJECT
245 1493 CDC914          CALL  PGNS
246 1496 C9              RET
247
248
249
250 1497 3E8D            EJECT  MVI  A, :8D
251 1499 CD7701          CALL  CPNT      RETURN CARRIAGE
252 149C 3ABC13          LDA  LINZ      GET REMAINING LINE COUNT
253 149F C603            ADI   3
254 14A1 47              MOV  B, A
255 14A2 3E8A            MVI  A, :8A
256 14A4 CD7701          EJ2   CALL  CPNT
257 14A7 05              DCR   B
258 14A8 C2A414          JNZ  EJ2
259 14AB 0648            MVI  B, 72
260 14AD 3EAD            MVI  A, '-'
261 14AF CD7701          EJ3   CALL  CPNT      PERFORATE PAGE
262 14B2 05              DCR   B
263 14B3 C2AF14          JNZ  EJ3

```



```

264 14B6 3E8D          MVI  A,:8D
265 14B8 CD7701        CALL CPNT
266 14BB 3E8A          MVI  A,:8A
267 14BD CD7701        CALL CPNT
268 14C0 CD7701        EJ4  CALL CPNT
269 14C3 3E3C          MVI  A,60
270 14C5 32BC13        STA  LINZ      INITIALIZE LINE COUNT
271 14C8 C9            RET
272
273 *
274 * PRINT AND UPDATE PAGE NUMBER
275 *
275 14C9 3ABB13        PGN8  LDA  PAGE
276 14CC 3C            INR  A
277 14CD 32BB13        STA  PAGE
278 14D0 0600          MVI  B,0
279 14D2 D60A          PGN81 SUI  10      CONVERT TO DECIMAL
280 14D4 FADB14        JM   PGN82
281 14D7 04            INR  B
282 14D8 C3D214        JMP  PGN81
283 14DB C6BA          PGN82 ADI  :BA
284 14DD 32C313        STA  PD2
285 14E0 78            MOV  A,B
286 14E1 C6B0          ADI  :B0
287 14E3 32C213        STA  PD1
288 14E6 21BD13        LXI  H,PGNO
289 14E9 CD6C14        CALL LPNT
290 14EC 2B            DCX  H
291 14ED CD6C14        CALL LPNT
292 14F0 C9            RET
293 PDVC EQU 0
294 14F1 21FA14        NOPROG LXI  H,NPMSG
295 14F4 CD9901        CALL PRT
296 14F7 C31901        JMP  MON2
297 14FA CECFA0D0      NPMSG  ASC  'NO PROGRAM IN MEMORY'
298 150E 8D            DATA :8D
299
300 *
301 * OPERAND FIELD ANALYSIS SECTION
302 *
303 * ENTER WITH ADDRESS OF 1ST CHARACTER IN D&E
304 * EXIT WITH TERMINATING CHARACTER IN A, FIELD
305 * CLASS(ES) IN B, FIELD VALUE IN H&L.
306 * UPON RETURN D&E CONTAIN THE ADDRESS OF THE CHAR-
307 * ACTER AFTER THE ONE CAUSING TERMINATION OF THE
308 * SCAN. CARRY RETURNS OFF IF THE RESULT
309 * CAN BE HELD IN 8 BITS, ON IF IT CANNOT
310 *
311 *
312 * FIELD CLASSES:
313 *
314 * -2 LENGTH ERROR. NUMERICAL FIELD RESULTS
315 * IN NUMBER WHICH CANNOT BE HELD IN 16 BITS.
316 *
317 * -1 UNDEFINED SYMBOL OR UNDEFINED FIELD TYPE.
318 *
319 * 0 NULL FIELD. FIRST CHARACTER IS BLANK OR
320 * COMMA.
321 *

```

```

322      * 1      NOT USED
323      *
324      * 2      HEXADECIMAL FIELD
325      *
326      * 4      DECIMAL FIELD
327      *
328      * 8      OCTAL FIELD
329      *
330      * 16     SYMBOLIC FIELD
331      *
332      * 32     ASCII STRING. FOR THIS FIELD TYPE THE VALUE
333      *         RETURNED IN H&L IS THE LENGTH OF THE STRING
334      *
335      *         THE CLASS RETURNED IN B IS THE LOGICAL SUM OF
336      *         THE CLASSES OF THE COMPONENT FIELDS OF THE ENTIRE
337      *         STRING.
338      *
339      *
340      *
341 150F AF      FANLYZ  ZAR
342 1510 32D316 STA      FTYPE      SET TYPE TO NULL
343 1513 3EAB   MVI      A,'+'
344 1515 210000 LXI      H,0      CLEAR RESULT AREA
345 1518 22D716 FANI     SHLD     FNRSLT  STORE RESULT
346 151B 210000 FANIA    LXI      H,0      CLEAR PREVIOUS RESULT
347 151E FEAB   CPI      '+'     CHECK FOR OPERATOR
348 1520 CA2815 JEQ      FAN2
349 1523 FEAD   CPI      '-'
350 1525 C23015 JNE      FAN3
351 1528 32D416 FAN2    STA      FANOP      SET OPERATOR
352 152B 1A     LDAX     D
353 152C 13     INX      D
354 152D C31B15 JMP      FANIA
355 1530 FE00   FAN3    CPI      ' '      BLANK?
356 1532 CA9116 JEQ      FANEND    YES, EXIT
357 1535 FEAC   CPI      ','      COMMA
358 1537 CA9116 JEQ      FANEND    YES, EXIT
359 153A FE8D   CPI      :8D      CARRIAGE RETURN?
360 153C CA9116 JEQ      FANEND
361 153F FEAF   CPI      :A7      APOSTROPHE?
362 1541 CAB115 JEQ      FASC      ASCII FIELD
363 1544 FEC0   CPI      '@'      AT SIGN?
364 1546 CAF015 JEQ      FOCT      OCTAL FIELD
365 1549 FEBA   CPI      ':'      COLON
366 154B CA5716 JEQ      FHXDC     HEXADECIMAL FIELD
367 154E FEB0   CPI      '0'      TEST FOR DECIMAL
368 1550 DACF16 JAL      FUNDEF    UNDEFINED IF LESS
369 1553 FEBA   CPI      :BA
370 1555 DA1D16 JAL      FDECM    DECIMAL FIELD
371 1558 FEC1   CPI      'A'
372 155A DACF16 JAL      FUNDEF    CHECK FOR SYMBOL
373 155D FEDB   CPI      'I'
374 155F D2CF16 JGE      FUNDEF
375      *
376      * SYMBOLIC FIELD
377      *
378 1562 1B     DCX      D
379 1563 21A0A0 LXI      H,:A0A0      BLANK COPY BUFFER

```

|                 |               |      |          |                              |
|-----------------|---------------|------|----------|------------------------------|
| 380 1566 22D916 |               | SHLD | FBUF     |                              |
| 381 1569 22DB16 |               | SHLD | FBUF+2   |                              |
| 382 156C 22DD16 |               | SHLD | FBUF+4   |                              |
| 383 156F 21D916 |               | LXI  | H, FBUF  | ADDRESS TO INDEX             |
| 384 1572 0606   |               | MVI  | B, 6     | CHARACTER COUNT              |
| 385 1574 1A     | FSYM1         | LDAX | D        |                              |
| 386 1575 13     |               | INX  | D        |                              |
| 387 1576 CDAD16 |               | CALL | ECHECK   | CHECK FOR TERMINATOR         |
| 388 1579 DA8A15 |               | JC   | FANE05   |                              |
| 389 157C 77     |               | MOV  | M, A     |                              |
| 390 157D 23     |               | INX  | H        |                              |
| 391 157E 05     |               | DCR  | B        |                              |
| 392 157F C27415 |               | JNZ  | FSYM1    |                              |
| 393 1582 1A     |               | LDAX | D        | GET 7TH CHARACTER            |
| 394 1583 13     |               | INX  | D        |                              |
| 395 1584 CDAD16 |               | CALL | ECHECK   | CHECK FOR TERMINATOR         |
| 396 1587 D2CF16 |               | JNC  | FUNDEF   | BAD SYMBOL IF NOT TERMINATOR |
| 397 158A F5     | FANE05        | PUSH | PSW      | SAVE LAST CHARACTER          |
| 398 158B 21D916 |               | LXI  | H, FBUF  | BUFFER ADDRESS TO INDEX      |
| 399 158E CDB011 |               | CALL | SRCSTB   | SEARCH THE SYMBOL TABLE      |
| 400 1591 DACE16 |               | JC   | FUNDEF-1 | UNDEFINED SYMBOL             |
| 401 1594 3AD316 |               | LDA  | FTYPE    | OTHERWISE SET TYPE           |
| 402 1597 F610   |               | ORI  | FSYMBT   |                              |
| 403 1599 32D316 | FANS          | STA  | FTYPE    |                              |
| 404 159C 3AD416 | FANEOL        | LDA  | FANOP    | GET OPERATOR                 |
| 405 159F FEAD   |               | CPI  | '-'      |                              |
| 406 15A1 C2A715 |               | JNE  | FANE2    |                              |
| 407 15A4 CD3304 |               | CALL | NEGHL    |                              |
| 408 15A7 44     | FANE2         | MOV  | B, H     |                              |
| 409 15A8 4D     |               | MOV  | C, L     |                              |
| 410 15A9 2AD716 |               | LHLD | FNRSLT   | GET PREVIOUS RESULT          |
| 411 15AC 09     |               | DAD  | B        | COMBINE WITH NEW RESULT      |
| 412 15AD F1     |               | POP  | PSW      | RETRIEVE TERMINATOR          |
| 413 15AE C31815 |               | JMP  | FANI     | AND GET NEXT COMPONENT       |
| 414             | *             |      |          |                              |
| 415             | * ASCII FIELD |      |          |                              |
| 416             | *             |      |          |                              |
| 417 15B1 1A     | FASC          | LDAX | D        |                              |
| 418 15B2 010000 |               | LXI  | B, 0     |                              |
| 419 15B5 1A     | FASC1         | LDAX | D        |                              |
| 420 15B6 13     |               | INX  | D        |                              |
| 421 15B7 FEAF   |               | CPI  | :A7      | SECOND APOSTROPHE?           |
| 422 15B9 CAC715 |               | JEQ  | FASC2    | YES, EXIT                    |
| 423 15BC FE8D   |               | CPI  | :8D      | C.R. IN STRING?              |
| 424 15BE CACF16 |               | JEQ  | FUNDEF   | YES, GARBAGE                 |
| 425 15C1 48     |               | MOV  | C, B     |                              |
| 426 15C2 47     |               | MOV  | B, A     |                              |
| 427 15C3 23     |               | INX  | H        | BUMP FIELD COUNT             |
| 428 15C4 C3B515 |               | JMP  | FASC1    |                              |
| 429 15C7 1A     | FASC2         | LDAX | D        | GET CHARACTER AFTER TERMINAT |
| 430 15C8 13     |               | INX  | D        |                              |
| 431 15C9 FE80   |               | CPI  | ' '      | CHECK FOR LEGAL TERMINATOR   |
| 432 15CB CADB15 |               | JEQ  | FASC3    |                              |
| 433 15CE FEAC   |               | CPI  | ' '      | ONLY BLANK, COMMA OR         |
| 434 15D0 CADB15 |               | JEQ  | FASC3    |                              |
| 435 15D3 FE8D   |               | CPI  | :8D      | CARRIAGE RETURN ALLOWED HERE |
| 436 15D5 CADB15 |               | JEQ  | FASC3    |                              |
| 437 15D8 C3CF16 |               | JMP  | FUNDEF   | OTHERWISE GARBAGE            |

|                 |                 |      |        |                           |
|-----------------|-----------------|------|--------|---------------------------|
| 438 1508 F5     | FASC3           | PUSH | PSW    |                           |
| 439 150C 79     |                 | MOV  | A,C    |                           |
| 440 15DD 32D616 |                 | STA  | NLACHR |                           |
| 441 15E0 78     |                 | MOV  | A,B    |                           |
| 442 15E1 32D516 |                 | STA  | LACHAR |                           |
| 443 15E4 3AD316 |                 | LDA  | FTYPE  |                           |
| 444 15E7 F620   |                 | ORI  | FASCBT |                           |
| 445 15E9 32D316 |                 | STA  | FTYPE  |                           |
| 446 15EC 47     |                 | MOV  | B,A    | CLASS TO B                |
| 447 15ED F1     |                 | POP  | PSW    | TERMINATOR TO A           |
| 448 15EE B7     |                 | RSC  |        |                           |
| 449 15EF C9     |                 | RET  |        |                           |
| 450             |                 |      |        |                           |
| 451             | *               |      |        |                           |
| 452             | * OCTAL FIELD   |      |        |                           |
| 453 15F0 0600   | FOCT            | MVI  | B,0    | CLEAR UPPER WORK REGISTER |
| 454 15F2 1A     |                 | LDAX | D      |                           |
| 455 15F3 13     |                 | INX  | D      |                           |
| 456 15F4 CDAD16 |                 | CALL | ECHECK |                           |
| 457 15F7 DA1416 |                 | JC   | FOEND  |                           |
| 458 15FA FEB0   |                 | CPI  | '0'    | CHECK LEGITIMATE OCTAL    |
| 459 15FC DACF16 |                 | JAL  | FUNDEF |                           |
| 460 15FF FEB8   |                 | CPI  | '8'    |                           |
| 461 1601 D2CF16 |                 | JGE  | FUNDEF |                           |
| 462 1604 E607   |                 | ANI  | 7      |                           |
| 463 1606 4F     |                 | MOV  | C,A    | SAVE INCOMING DIGIT       |
| 464 1607 7C     |                 | MOV  | A,H    | CHECK FOR OVERFLOW        |
| 465 1608 FE20   |                 | CPI  | 32     |                           |
| 466 160A D2CB16 |                 | JGE  | FLNGTH | LENGTH ERROR              |
| 467 160D 29     |                 | DAD  | H      | SHIFT LEFT 3              |
| 468 160E 29     |                 | DAD  | H      |                           |
| 469 160F 29     |                 | DAD  | H      |                           |
| 470 1610 09     |                 | DAD  | B      | MERGE INCOMING DIGIT      |
| 471 1611 C3F015 |                 | JMP  | FOCT   |                           |
| 472 1614 F5     | FOEND           | PUSH | PSW    |                           |
| 473 1615 3AD316 |                 | LDA  | FTYPE  |                           |
| 474 1618 F608   |                 | ORI  | FOCTBT |                           |
| 475 161A C39915 |                 | JMP  | FANS   |                           |
| 476             |                 |      |        |                           |
| 477             | *               |      |        |                           |
| 478             | * DECIMAL FIELD |      |        |                           |
| 479 161D 1B     | FDECML          | DCX  | D      |                           |
| 480 161E 1A     | FDC2            | LDAX | D      |                           |
| 481 161F 13     |                 | INX  | D      |                           |
| 482 1620 CDAD16 |                 | CALL | ECHECK |                           |
| 483 1623 DA4E16 |                 | JC   | FDCEND |                           |
| 484 1626 FEB0   |                 | CPI  | '0'    |                           |
| 485 1628 DACF16 |                 | JAL  | FUNDEF |                           |
| 486 162B FEB8   |                 | CPI  | :BA    |                           |
| 487 162D D2CF16 |                 | JGE  | FUNDEF |                           |
| 488 1630 E60F   |                 | ANI  | :F     | MASK OUT ASCII BITS       |
| 489 1632 29     |                 | DAD  | H      | PREV. RESULT TIMES 10     |
| 490 1633 DACB16 |                 | JC   | FLNGTH |                           |
| 491 1636 44     |                 | MOV  | B,H    |                           |
| 492 1637 4D     |                 | MOV  | C,L    |                           |
| 493 1638 29     |                 | DAD  | H      |                           |
| 494 1639 DACB16 |                 | JC   | FLNGTH |                           |
| 495 163C 29     |                 | DAD  | H      |                           |

|     |      |        |        |                   |                     |
|-----|------|--------|--------|-------------------|---------------------|
| 496 | 163D | DACB16 | JC     | FLNGTH            |                     |
| 497 | 1640 | 09     | DAD    | B                 |                     |
| 498 | 1641 | DACB16 | JC     | FLNGTH            |                     |
| 499 | 1644 | 4F     | MOV    | C,A               |                     |
| 500 | 1645 | 0600   | MVI    | B,0               |                     |
| 501 | 1647 | 09     | DAD    | B                 | ADD NEW DIGIT       |
| 502 | 1648 | DACB16 | JC     | FLNGTH            |                     |
| 503 | 164B | C31E16 | JMP    | FDC2              |                     |
| 504 | 164E | F5     | FDCEND | PUSH              | PSW                 |
| 505 | 164F | 3AD316 | LDA    | FTYPE             |                     |
| 506 | 1652 | F604   | ORI    | FDECBT            |                     |
| 507 | 1654 | C39915 | JMP    | FAN8              |                     |
| 508 |      |        |        |                   |                     |
| 509 |      |        | *      | HEXADECIMAL FIELD |                     |
| 510 |      |        | *      |                   |                     |
| 511 | 1657 | 0600   | FHXDC  | MVI               | B,0                 |
| 512 | 1659 | 1A     | FHXDC0 | LDAX              | D                   |
| 513 | 165A | 13     |        | INX               | D                   |
| 514 | 165B | CDAD16 |        | CALL              | ECHECK              |
| 515 | 165E | DA8816 | JC     | FHXEND            |                     |
| 516 | 1661 | FEB0   | CPI    | '0'               | CHECK FOR LEGAL HEX |
| 517 | 1663 | DACF16 | JAL    | FUNDEF            |                     |
| 518 | 1666 | FEB8   | CPI    | :BA               |                     |
| 519 | 1668 | DA7716 | JAL    | FHXOK             |                     |
| 520 | 166B | FEC1   | CPI    | 'A'               |                     |
| 521 | 166D | DACF16 | JAL    | FUNDEF            |                     |
| 522 | 1670 | FEC7   | CPI    | 'G'               |                     |
| 523 | 1672 | D2CF16 | JGE    | FUNDEF            |                     |
| 524 | 1675 | D607   | SUI    | 7                 | BIAS CHARACTER DOWN |
| 525 | 1677 | E60F   | FHXOK  | ANI               | :F                  |
| 526 | 1679 | 4F     |        | MOV               | C,A                 |
| 527 | 167A | 7C     |        | MOV               | A,H                 |
| 528 | 167B | FE10   |        | CPI               | 16                  |
| 529 | 167D | D2CB16 | JGE    | FLNGTH            | CHECK FOR OVERFLOW  |
| 530 | 1680 | 29     | DAD    | H                 |                     |
| 531 | 1681 | 29     | DAD    | H                 |                     |
| 532 | 1682 | 29     | DAD    | H                 |                     |
| 533 | 1683 | 29     | DAD    | H                 |                     |
| 534 | 1684 | 09     | DAD    | B                 | MERGE NEW DIGIT     |
| 535 | 1685 | C35916 | JMP    | FHXDC0            |                     |
| 536 | 1688 | F5     | FHXEND | PUSH              | PSW                 |
| 537 | 1689 | 3AD316 | LDA    | FTYPE             |                     |
| 538 | 168C | F602   | ORI    | FHEXBT            |                     |
| 539 | 168E | C39915 | JMP    | FAN8              |                     |
| 540 |      |        |        |                   |                     |
| 541 |      |        | *      | EXIT POINT        |                     |
| 542 |      |        | *      |                   |                     |
| 543 | 1691 | 2AD716 | FANEND | LHLD              | FNRSLT              |
| 544 | 1694 | 32D916 |        | STA               | FBUF                |
| 545 | 1697 | 3AD316 |        | LDA               | FTYPE               |
| 546 | 169A | 47     |        | MOV               | B,A                 |
| 547 | 169B | 7C     |        | MOV               | A,H                 |
| 548 | 169C | B7     |        | ORA               | A                   |
| 549 | 169D | CAA516 |        | JZ                | FEND                |
| 550 | 16A0 | 3C     |        | INR               | A                   |
| 551 | 16A1 | C2A916 |        | JNZ               | FSTC                |
| 552 | 16A4 | B7     |        | RSC               |                     |
| 553 | 16A5 | 3AD916 | FEND   | LDA               | FBUF                |
|     |      |        |        |                   | OK FOR 8 BITS       |



```

554 16A8 C9          RET
555 16A9 37          FSTC  STC
556 16AA C3A516      JMP  FEND
557                  *
558                  * CHECK FOR TERMINATOR
559                  *
560 16AD FE80          ECHECK CPI  '/'
561 16AF CAC816        JEQ  ECH2
562 16B2 FEAC          CPI  '/'
563 16B4 CAC916        JEQ  ECHA
564 16B7 FE8D          CPI  :8D
565 16B9 CAC816        JEQ  ECH2
566 16BC FEAB          CPI  '+'
567 16BE CAC816        JEQ  ECH2
568 16C1 FEAD          CPI  '-'
569 16C3 CAC816        JEQ  ECH2
570 16C6 B7           RSC
571 16C7 C9           RET
572 16C8 1B           ECH2 DCX  D
573 16C9 37           ECHA  STC
574 16CA C9           RET
575                  *
576                  * LENGTH ERROR
577                  *
578 16CB 06FE          FLNGTH MVI  B,-2
579 16CD C9           RET
580                  *
581                  * UNDEFINED SYMBOL OR FIELD TYPE
582                  *
583 16CE F1            POP  PSW
584 16CF 06FF          FUNDEF MVI  B,-1
585 16D1 B7           RSC
586 16D2 C9           RET
587                  *
588                  * WORK SPACE
589                  *
590 16D3 00            FTYPE ***
591 16D4 00            FANOP ***
592 16D5 00            LACHR ***
593 16D6 00            NLACHR ***
594 16D7 0000          FNRSLT DBL  0
595 16D9              FBUF  RES  6
596              FHXB T EQU  2
597              FDECB T EQU  4
598              FOCTB T EQU  8
599              FSymb T EQU 16
600              FASCB T EQU 32
601                  *
602                  * CHECK LABEL VALIDITY
603                  *
604 16DF E5            SLLBL  PUSH  H
605 16E0 C5            PUSH  B
606 16E1 219D07        LXI   H,LABEL
607 16E4 7E            MOV   A,M
608 16E5 FEC1          CPI   'A'
609 16E7 DA0B17        JAL   SLL3
610 16EA FEDB          CPI   'I'
611 16EC D20B17        JGE   SLL3

```

|     |             |      |      |         |                         |
|-----|-------------|------|------|---------|-------------------------|
| 612 | 16EF 23     |      | INX  | H       |                         |
| 613 | 16F0 0605   |      | MVI  | B,5     |                         |
| 614 | 16F2 7E     | SLL2 | MOV  | A,M     |                         |
| 615 | 16F3 FEAB   |      | CPI  | '+'     |                         |
| 616 | 16F5 CA0B17 |      | JEQ  | SLL3    |                         |
| 617 | 16F8 FEAD   |      | CPI  | '-'     |                         |
| 618 | 16FA CA0B17 |      | JEQ  | SLL3    |                         |
| 619 | 16FD FEAC   |      | CPI  | '/'     |                         |
| 620 | 16FF CA0B17 |      | JEQ  | SLL3    |                         |
| 621 | 1702 23     |      | INX  | H       |                         |
| 622 | 1703 05     |      | DCR  | B       |                         |
| 623 | 1704 C2F216 |      | JNZ  | SLL2    |                         |
| 624 | 1707 B7     |      | RSC  |         |                         |
| 625 | 1708 C1     |      | POP  | B       |                         |
| 626 | 1709 E1     |      | POP  | H       |                         |
| 627 | 170A C9     |      | RET  |         |                         |
| 628 | 170B 37     | SLL3 | STC  |         |                         |
| 629 | 170C C1     |      | POP  | B       |                         |
| 630 | 170D E1     |      | POP  | H       |                         |
| 631 | 170E C9     |      | RET  |         |                         |
| 632 |             |      |      |         |                         |
| 633 |             |      |      |         |                         |
| 634 |             |      |      |         |                         |
| 635 | 170F 210000 |      | LXI  | H,0     |                         |
| 636 | 1712 22AD13 |      | SHLD | NSYMS   | CLEAR SYMBOL COUNT      |
| 637 | 1715 22B818 |      | SHLD | ILC     | AND LOCATION COUNTER    |
| 638 | 1718 22BA18 |      | SHLD | ORGVAL  | SET ORIGIN TO ZERO      |
| 639 | 171B 22BE18 |      | SHLD | TLFLAG  | CLEAR TITLE FLAG        |
| 640 | 171E 22AC22 |      | SHLD | P2LNO   |                         |
| 641 | 1721 CD3714 | PIB  | CALL | FSRCD   | GET SOURCE RECORD       |
| 642 | 1724 CDA019 |      | CALL | CNVLNO  |                         |
| 643 | 1727 2AB818 |      | LHLD | ILC     | LOCATION COUNTER TO H&L |
| 644 | 172A 3A9D07 |      | LDA  | LABEL   | CHECK FOR COMMENT       |
| 645 | 172D FEAA   |      | CPI  | '*'     |                         |
| 646 | 172F CA2117 |      | JEQ  | PIB     |                         |
| 647 | 1732 3A8407 |      | LDA  | CLASS   | CHECK FOR EQU           |
| 648 | 1735 B7     |      | ORA  | A       |                         |
| 649 | 1736 C27517 |      | JNZ  | PIC     |                         |
| 650 | 1739 3A8507 |      | LDA  | SKEL    |                         |
| 651 | 173C FE07   |      | CPI  | 7       |                         |
| 652 | 173E C27517 |      | JNE  | PIC     |                         |
| 653 | 1741 CDDF16 |      | CALL | SLLBL   |                         |
| 654 | 1744 DA5619 |      | JC   | PIEQR   |                         |
| 655 | 1747 11AB07 |      | LXI  | D,OPNDF |                         |
| 656 | 174A CD0F15 |      | CALL | FANLYZ  | GET EQU FIELD VALUE     |
| 657 | 174D FE8D   |      | CPI  | :8D     | END IN C.R. OR BLANK    |
| 658 | 174F CA5717 |      | JEQ  | PIB1    |                         |
| 659 | 1752 FE00   |      | CPI  | ' '     |                         |
| 660 | 1754 C22117 |      | JNE  | PIB     |                         |
| 661 | 1757 78     | PIB1 | MOV  | A,B     |                         |
| 662 | 1758 FE02   |      | CPI  | 2       |                         |
| 663 | 175A FA2E19 |      | JC   | PIERFL  |                         |
| 664 | 175D FE20   |      | CPI  | FASCBT  |                         |
| 665 | 175F DA7517 |      | JAL  | PIC     |                         |
| 666 | 1762 CA6817 |      | JEQ  | PIB2    |                         |
| 667 | 1765 D22E19 |      | JGE  | PIERFL  |                         |
| 668 | 1768 7D     | PIB2 | MOV  | A,L     |                         |
| 669 | 1769 B7     |      | ORA  | A       |                         |

|     |      |        |      |                   |                                |
|-----|------|--------|------|-------------------|--------------------------------|
| 670 | 176A | FA2117 | JM   | PIB               |                                |
| 671 | 176D | FE03   | CPI  | 3                 |                                |
| 672 | 176F | D22117 | JGE  | PIB               |                                |
| 673 | 1772 | 2AD516 | LHLD | LACHAR            |                                |
| 674 | 1775 | 3A9D07 | LDA  | LABEL             | PIC WAS THERE A LABEL?         |
| 675 | 1778 | FEA0   | CPI  | ' '               |                                |
| 676 | 177A | CA9F17 | JEQ  | PIE               |                                |
| 677 | 177D | CDDF16 | CALL | SLLBL             |                                |
| 678 | 1780 | DA7E19 | JC   | PILERR            |                                |
| 679 | 1783 | 22BC18 | SHLD | PITMP             | SAVE SYMBOL VALUE              |
| 680 | 1786 | 219D07 | LXI  | H,LABEL           |                                |
| 681 | 1789 | CDB011 | CALL | SRCSTB            | SYMBOL ALREADY IN TABLE?       |
| 682 | 178C | DA9917 | JC   | PID               | NO                             |
| 683 | 178F | 2AA513 | LHLD | STLOC             | YES, MARK IT DOUBLE            |
| 684 | 1792 | 7E     | MOV  | A,M               | DEFINED                        |
| 685 | 1793 | E67F   | ANI  | :7F               |                                |
| 686 | 1795 | 77     | MOV  | M,A               | AND REPLACE IT                 |
| 687 | 1796 | C39F17 | JMP  | PIE               |                                |
| 688 | 1799 | 2ABC18 | LHLD | PITMP             |                                |
| 689 | 179C | CD7111 | CALL | SYMENT            | PID ENTER SYMBOL IN TABLE      |
| 690 | 179F | 3A8407 | LDA  | CLASS             |                                |
| 691 | 17A2 | FEFF   | CPI  | :FF               |                                |
| 692 | 17A4 | CAB218 | JEQ  | PIINC4            |                                |
| 693 | 17A7 | E67F   | ANI  | :7F               |                                |
| 694 | 17A9 | CAC017 | JZ   | PIC0              |                                |
| 695 | 17AC | CDD918 | CALL | CKZ80             | CHECK FOR AMBIGUOUS INST'S     |
| 696 | 17AF | 1600   | MVI  | D,0               |                                |
| 697 | 17B1 | 21BF18 | LXI  | H,PILTBL-1        | ADDRESS OF LENGTH TABLE        |
| 698 | 17B4 | 19     | DAD  | D                 |                                |
| 699 | 17B5 | 5E     | MOV  | E,M               |                                |
| 700 | 17B6 | 2AB818 | LHLD | ILC               | PISUM GET OLD LOCATION COUNTER |
| 701 | 17B9 | 19     | DAD  | D                 | ADD NEW LENGTH                 |
| 702 | 17BA | 22B818 | SHLD | ILC               | RESTORE LOCATION COUNTER       |
| 703 | 17BD | C32117 | JMP  | PIB               | AND GET NEXT INSTRUCTION       |
| 704 |      |        |      |                   |                                |
| 705 |      |        | *    | CLASS 0 PROCESSOR |                                |
| 706 |      |        | *    |                   |                                |
| 707 | 17C0 | 11AB07 | LXI  | D,OPNDF           |                                |
| 708 | 17C3 | 3A8507 | LDA  | SKEL              | GET INSTRUCTION SKELETON       |
| 709 | 17C6 | FE00   | CPI  | 0                 |                                |
| 710 | 17C8 | CA0E18 | JEQ  | PIORG             | ORG                            |
| 711 | 17CB | FE01   | CPI  | 1                 |                                |
| 712 | 17CD | CA0518 | JEQ  | HANGUP            |                                |
| 713 | 17D0 | FE02   | CPI  | 2                 |                                |
| 714 | 17D2 | CARC18 | JEQ  | PIINC3            | ARS                            |
| 715 | 17D5 | FE03   | CPI  | 3                 |                                |
| 716 | 17D7 | CA2718 | JEQ  | PITITL            | TITL                           |
| 717 | 17DA | FE04   | CPI  | 4                 |                                |
| 718 | 17DC | CA2117 | JEQ  | PIB               | UNL                            |
| 719 | 17DF | FE05   | CPI  | 5                 |                                |
| 720 | 17E1 | CA2117 | JEQ  | PIB               | LST                            |
| 721 | 17E4 | FE06   | CPI  | 6                 |                                |
| 722 | 17E6 | CA6D18 | JEQ  | PIDTA             | DATA                           |
| 723 | 17E9 | FE07   | CPI  | 7                 |                                |
| 724 | 17EB | CA2117 | JEQ  | PIB               | EQU                            |
| 725 | 17EE | FE08   | CPI  | 8                 |                                |
| 726 | 17F0 | CA5118 | JEQ  | PIASC             | ASC                            |
| 727 | 17F3 | FE09   | CPI  | 9                 |                                |

```

728 17F5 CA3F18      JEQ  PIRE5      RES
729 17F8 FE0A        CPI  10
730 17FA CA2117      JEQ  PIB        REM
731 17FD FE0B        CPI  11
732 17FF CA9618      JEQ  PIINC2     DBL
733 1802 C3B218      JMP  PIINC4     JUST IN CASE ...
734 1805 3A9F13      HANGUP LDA  INFLAG  TEST FOR TAPE INPUT
735 1808 B7          ORA  A
736 1809 C8          RZ
737 180A CD7401      CALL SB        MEMORY INPUT
738 180D C9          RET          WAIT FOR SPACE BAR
739
740      * ORG PROCESSOR
741
742 180E CD0F15      PIORG  CALL  FANLYZ  GET OPERAND VALUE
743 1811 FEA0        CPI  ' '      MUST END IN BLANK
744 1813 CA1B18      JEQ  PIORG2
745 1816 FE8D        CPI  -8D      OR CARRIAGE RETURN
746 1818 C22117      JNE  PIB      IGNORE ORG
747 181B CDE51F      PIORG2 CALL  TCHK1
748 181E DA2E19      JC   PIERFL
749 1821 22B818      SHLD ILC      SET NEW LOCATION COUNTER
750 1824 C32117      JMP  PIB      GET NEXT INSTRUCTION
751
752      * TITL PROCESSOR
753
754 1827 0626      PITITL MWI  B,38      MOVE TITLE FIELD
755 1829 21AB07      LXI  H,OPNDF
756 182C 11C813      LXI  D,TITLE
757 182F 7E        PITTL2 MOV  A,M
758 1830 12        STAX D
759 1831 23        INX  H
760 1832 13        INX  D
761 1833 05        DCR  B
762 1834 C22F18      JNZ  PITTL2
763 1837 3EFF      MVI  A,-1
764 1839 32BE18      STA  TLFLAG
765 183C C32117      JMP  PIB
766
767      * RES PROCESSOR
768
769 183F CD0F15      PIRE5  CALL  FANLYZ  GET BLOCK SIZE
770 1842 78        MOV  A,B      CHECK FIELD TYPE
771 1843 FE02      CPI  2
772 1845 FA2E19      JC   PIERFL
773 1848 FE20      CPI  32
774 184A D22E19      JGE  PIERFL
775 184D EB        PIRE52 XCHG
776 184E C3B617      JMP  PISUM
777
778      * ASC PROCESSOR
779
780 1851 CD0F15      PIASC  CALL  FANLYZ
781 1854 FEA0      CPI  ' '      MUST END IN BLANK
782 1856 CA5E18      JEQ  PIASC2
783 1859 FE8D      CPI  -8D      OR CARRIAGE RETURN
784 185B C2A018      JNE  PIINC1
785 185E 78        PIASC2 MOV  A,B      CHECK TYPE

```

```

786 185F FE20          CPI 32
787 1861 C2A018        JNE PIINC1
788 1864 7D           MOV A,L
789 1865 FE25          CPI 37
790 1867 D2A018        JGE PIINC1
791 186A C34D18        JMP PIRES2
792
793 * DATA PROCESSOR
794 *
795 186D 210000        PIDTA LXI H,0
796 1870 22BC18        SHLD PITMP
797 1873 CD0F15        PIDTA1 CALL FANLYZ
798 1876 DA9A18        JC PIDTA3
799 1879 4F           MOV C,A
800 187A 78           MOV A,B      CHECK TYPE FIRST
801 187B FE02          CPI 2
802 187D FA9A18        JC PIDTA3
803 1880 FE20          CPI 32
804 1882 DA8B18        JAL PIDTA2
805 1885 7D           MOV A,L
806 1886 FE01          CPI 1      CHECK CORRECT FIELD LENGTH
807 1888 C29A18        JNE PIDTA3
808 188B E5           PIDTA2 PUSH H
809 188C 2ABC18        LHLD PITMP
810 188F 23           INX H      INCREMENT LENGTH
811 1890 22BC18        SHLD PITMP
812 1893 E1           POP H
813 1894 79           MOV A,C
814 1895 FEAC          CPI ','   LAST ITEM?
815 1897 CA7318        JEQ PIDTA1
816 189A 2ABC18        PIDTA3 LHLD PITMP
817 189D C34D18        JMP PIRES2
818
819 18A0 210100        PIINC1 LXI H,1
820 18A3 C34D18        JMP PIRES2
821 18A6 210200        PIINC2 LXI H,2
822 18A9 C34D18        JMP PIRES2
823 18AC 210300        PIINC3 LXI H,3
824 18AF C34D18        JMP PIRES2
825 18B2 210400        PIINC4 LXI H,4
826 18B5 C34D18        JMP PIRES2
827 18B8 0000        ILC DBL 0
828 18BA 0000        ORGVAL DBL 0
829 18BC 0000        PITMP DBL 0
830 18BE 0000        TLFLAG DBL 0
831
832 * INSTRUCTION LENGTH TABLE
833 *
834
835
836
837 18C0 01          PILTBL DATA 1 1
838 18C1 01          DATA 1 2
839 18C2 02          DATA 2 3
840 18C3 03          DATA 3 4
841 18C4 03          DATA 3 5
842 18C5 01          DATA 1 6
843 18C6 01          DATA 1 7

```



|                 |                |                             |
|-----------------|----------------|-----------------------------|
| 844 18C7 02     | DATA 2         | 8                           |
| 845 18C8 02     | DATA 2         | 9                           |
| 846 18C9 01     | DATA 1         | A                           |
| 847             |                |                             |
| 848             |                |                             |
| 849             |                | Z80 INSTRUCTION TYPES       |
| 850 18CA 03     | DATA 3         | B                           |
| 851 18CB 04     | DATA 4         | C                           |
| 852 18CC 01     | DATA 1         | D                           |
| 853 18CD 02     | DATA 2         | E                           |
| 854 18CE 04     | DATA 4         | F                           |
| 855 18CF 04     | DATA 4         | 10                          |
| 856 18D0 02     | DATA 2         | 11                          |
| 857 18D1 02     | DATA 2         | 12                          |
| 858 18D2 02     | DATA 2         | 13                          |
| 859 18D3 02     | DATA 2         | 14                          |
| 860 18D4 02     | DATA 2         | 15                          |
| 861 18D5 04     | DATA 4         | 16                          |
| 862 18D6 04     | DATA 4         | 17                          |
| 863 18D7 02     | DATA 2         | 18                          |
| 864 18D8 02     | DATA 2         | 19                          |
| 865             |                |                             |
| 866             |                |                             |
| 867             |                |                             |
| 868             |                |                             |
| 869             |                |                             |
| 870             |                |                             |
| 871             |                |                             |
| 872 18D9 E61F   | CKZ80 ANI :1F  | CLEAR SUPERFLUOUS BITS      |
| 873 18DB 5F     | MOV E,A        | SAVE TYPE IN E              |
| 874 18DC FE05   | CPI 5          | IS IT AN LXI?               |
| 875 18DE C2FA18 | JNE C26        | NO, CHECK TYPE 6            |
| 876 18E1 3AAB07 | LDA OPNDF      | YES, GET 1ST OPERAND CHAR   |
| 877 18E4 FED8   | CPI 'X'        | CHECK FOR X OR Y            |
| 878 18E6 CAF018 | JEQ CKZ2       |                             |
| 879 18E9 FED9   | CPI 'Y'        |                             |
| 880 18EB CAF018 | JEQ CKZ2       |                             |
| 881 18EE 7B     | CKZRET MOV A,E | COPY TYPE TO A              |
| 882 18EF C9     | RET            |                             |
| 883 18F0 1E0F   | CKZ2 MVI E,:F  | CHANGE TO Z80 LXI X/Y       |
| 884 18F2 3E21   | MVI A,:21      | CHANGE INSTRUCTION SKELETON |
| 885 18F4 328507 | STA SKEL       |                             |
| 886 18F7 C3EE18 | JMP CKZRET     |                             |
| 887 18FA FE06   | C26 CPI 6      | CHECK FOR TYPE 6            |
| 888 18FC C0     | RNZ            | RETURN IF NOT               |
| 889 18FD 3A8507 | LDA SKEL       |                             |
| 890 1900 FE03   | CPI 3          | IS IT INX                   |
| 891 1902 CA1419 | JEQ C27        |                             |
| 892 1905 FE0B   | CPI :B         | OR DCX                      |
| 893 1907 CA1419 | JEQ C27        |                             |
| 894 190A FEC5   | CPI :C5        | OR PUSH/PSH                 |
| 895 190C CA1419 | JEQ C27        |                             |
| 896 190F FEC1   | CPI :C1        | OR POP                      |
| 897 1911 C2EE18 | JNE CKZRET     |                             |
| 898 1914 3AAB07 | C27 LDA OPNDF  | GET OPERAND CHARACTER       |
| 899 1917 FED8   | CPI 'X'        | CHECK FOR X OR Y            |
| 900 1919 CA2119 | JEQ C28        |                             |
| 901 191C FED9   | CPI 'Y'        |                             |

```

902 191E C2EE18      JNE  CKZRET
903 1921 3A8507      CZ8  LDA  SKEL
904 1924 F620        ORI  :20      CHANGE SKELETON FOR Z80 TYPE
905 1926 328507      STA  SKEL
906 1929 1E11        MVI  E, :11    CHANGE INSTRUCTION TYPE
907 192B C3EE18      JMP  CKZRET
908
909      *
910      * PASS 1 ERRORS
911 192E 218907      PIERFL LXI  H, HEX
912 1931 CD9901      CALL PRT
913 1934 215019      LXI  H, PIEMSG
914 1937 CD9901      CALL PRT
915 193A 3AA013      LDA  PRFLAG
916 193D B7          ORA  A
917 193E CA2117      JZ   PIB
918 1941 218907      LXI  H, HEX
919 1944 CD6C14      CALL LPNT
920 1947 215019      LXI  H, PIEMSG
921 194A CD6C14      CALL LPNT
922 194D C32117      JMP  PIB
923 1950 A0A0A0A0    PIEMSG ASC  ' P'
924 1955 8D          DATA :8D
925
926      *
927 1956 218907      PIEOUR LXI  H, HEX
928 1959 CD9901      CALL PRT
929 195C 217819      LXI  H, PIQMSG
930 195F CD9901      CALL PRT
931 1962 3AA013      LDA  PRFLAG
932 1965 B7          ORA  A
933 1966 CA2117      JZ   PIB
934 1969 218907      LXI  H, HEX
935 196C CD6C14      CALL LPNT
936 196F 217819      LXI  H, PIQMSG
937 1972 CD6C14      CALL LPNT
938 1975 C32117      JMP  PIB
939 1978 A0A0A0A0    PIQMSG ASC  ' S'
940 197D 8D          DATA :8D
941
942      *
943      * BAD SYMBOL, NOT EQU
944
945      *
946 197E 218907      PILERR LXI  H, HEX
947 1981 CD9901      CALL PRT
948 1984 217819      LXI  H, PIQMSG
949 1987 CD9901      CALL PRT
950 198A 3AA013      LDA  PRFLAG
951 198D B7          ORA  A
952 198E CA9F17      JZ   PIE
953 1991 218907      LXI  H, HEX
954 1994 CD6C14      CALL LPNT
955 1997 217819      LXI  H, PIQMSG
956 199A CD6C14      CALL LPNT
957 199D C39F17      JMP  PIE
958
959      *
960      * CONVERT LINE NUMBER
961
962 19A0 2AAC22      CNVLNO LHLD P2LNO
963 19A3 23          INX  H

```

```

960 19A4 22AC22      SHLD P2LNO
961 19A7 1118FC      LXI D,-1000
962 19AA CDDE03      CALL CNVDEC
963 19AD 119CFF      LXI D,-100
964 19B0 CDDE03      CALL CNVDEC
965 19B3 328907      STA HEX
966 19B6 11F6FF      LXI D,-10
967 19B9 CDDE03      CALL CNVDEC
968 19BC 328A07      STA HEX+1
969 19BF 7D          MOV A,L
970 19C0 F6B0        ORI :B0
971 19C2 328B07      STA HEX+2
972 19C5 C9          RET
973
974
975
*
* RESIDENT ASSEMBLER - PASS 2
*
PASS2 LXI H,0
      SHLD ILC          INITIALIZE FOR PAS
      SHLD ORGVAL
      SHLD P2LNO
      MVI A,60
      CALL PCHBLT
P2A   CALL DTCTCC
      JC RTMON
      CALL FSRCD
      CALL CNVLNO
      LXI H,0          CLEAR ERROR CODE WORD
                        AND OBJECT BUFFER
      SHLD ERRORS
      SHLD OCODE
      SHLD OCODE+2
      SHLD RSLNTH
      LXI H,:A0A0
                        CLEAR ERROR MESSAGE AREA
      SHLD ECA
      SHLD ECA+2
      SHLD ECA+4
      SHLD ECA+6
      LDA LABEL        CHECK FOR COMMENT
      CPI '*'
      JEQ P2PRC
      CPI '/'
                        WAS THERE A LABEL?
                        NO
      JEQ P2B
      CALL SLLBL
      JNC P2A1A
P2A1  LDA ERRORS
      ORI SYMBIT
      STA ERRORS
      JMP P2B
P2A1A LXI H,LABEL
      CALL SRCSTB      SEARCH TABLE FOR DUPLICATE
      LHLD STLOC
      MOV A,M
      ORA A
      JM P2B
      LDA ERRORS
      ORI MDFBIT      FLAG MULTIPLE DEFINITION
      STA ERRORS
P2B   LDA CLASS
      CPI :FF          ILLEGAL OPCODE?

```

```

018 1A3A CAE521      JEQ  P2BDOP
019 1A3D E67F        ANI  :7F
020 1A3F CA571A      JZ   P2C0
021 1A42 CDD918      CALL CKZ80
022 1A45 87          ADD  A
023 1A46 5F          MOV  E,A
024 1A47 1600        MVI  D,0
025 1A49 21031B      LXI  H,P2JTBL-2
026 1A4C 19          DAD  D
027 1A4D EB          XCHG
028 1A4E 1A          LDAX D
029 1A4F 6F          MOV  L,A
030 1A50 13          INX  D
031 1A51 1A          LDAX D
032 1A52 67          MOV  H,A
033 1A53 11AB07      LXI  D,OPNDF      OPERAND POINTER TO D-E
034 1A56 E9          PCHL      JUMP TO PROCESSOR
035
036
037
038 1A57 11AB07      * PSEUDO-OP PROCESSOR
039 1A5A 3A8507      *
040 1A5D FE00        P2C0 LXI  D,OPNDF
041 1A5F CAB01A      LDA  SKEL
042 1A62 FE01        CPI  0      ORG
043 1A64 CAD71A      JEQ  P2ORG
044 1A67 FE02        CPI  1      END
045 1A69 CAD91A      JEQ  P2END
046 1A6C FE03        CPI  2      ARS
047 1A6E CAD719      JEQ  P2ARS
048 1A71 FE04        CPI  3      TITL
049 1A73 CAF01A      JEQ  P2A
050 1A76 FE05        CPI  4      UNL
051 1A78 CAF71A      JEQ  P2UNL
052 1A7B FE06        CPI  5      LST
053 1A7D CA991B      JEQ  P2LST
054 1A80 FE07        CPI  6      DATA
055 1A82 CA9C1A      JEQ  P2DATA
056 1A85 FE08        CPI  7      EQU
057 1A87 CAF91B      JEQ  P2EQU
058 1A8A FE09        CPI  8      ASC
059 1A8C CA371B      JEQ  P2ASC
060 1A8F FE0A        CPI  9      RES
061 1A91 CA3F1C      JEQ  P2RES
062 1A94 FE0B        CPI  10     REM
063 1A96 CA4B1C      JEQ  P2REM
064 1A99 C3E521      CPI  11     DBL
065 1A9C 219D07      JEQ  P2DBL
066 1A9F CDB011      JMP  P2BDOP      JUST IN CASE...
067 1AA2 D23222      P2EQU LXI  H,LABEL
068 1AA5 3AE22        CALL SRCSTB
069 1AA8 F601        JNC  P2PRC
070 1AAA 32AE22      LDA  ERRORS
071 1AAD C33222      ORI  UNDFBT
072                STA  ERRORS
073                JMP  P2PRC
074
075 1AB0 CD0F15      * ORG PROCESSOR
                P2ORG  CALL  FANLYZ      GET OPERAND

```

```

076 IAB3 FE00      CPI      ' '      END IN BLANK OR C.R.
077 IAB5 CABD1A    JEQ      P2ORG2
078 IAB8 FE8D      CPI      :8D
079 IAB9 C2CC1A    JNE      P2ORG3
080 IABD CDE51F    P2ORG2 CALL   TCHK1
081 IAC0 DACC1A    JC       P2ORG3
082 IAC3 22B818    SHLD     ILC      SET NEW LOCATION COUNTER
083 IAC6 CD3C23    CALL     DP0BJR
084 IAC9 C33222    JMP      P2PRC
085 IACC 3AAE22    P2ORG3 LDA     ERRORS
086 IACF F601      ORI      UNDFBT
087 IAD1 32AE22    STA      ERRORS
088 IAD4 C33222    JMP      P2PRC
089
090 *
091 * END PROCESSOR
092
092 IAD7 C3F521    P2END   JMP     P2PRNT
093
094 *
095 * ARS PROCESSOR
096
096 IADA 3E03      P2ARS   MVI     A,3
097 IADC 32AF22    STA      P2LNTH
098 IADF 3E07      MVI     A,7
099 IAE1 32B022    STA      OCODE   SET OBJECT
100 IAE4 211F1F    LXI     H,:1F1F  RAR+RAR
101 IAE7 22B122    SHLD     OCODE+1
102 IAEA CD1223    CALL     PCHOBJ
103 IAD8 C3F521    JMP      P2PRNT
104
105 *
106 * UNL PROCESSOR
107
107 IAF0 AF        P2UNL   ZAR
108 IAF1 32A013    P2UNLA  STA     PRFLAG
109 IAF4 C3D719    JMP      P2A
110
111 *
112 * LST PROCESSOR
113
113 IAF7 3AA213    P2LST   LDA     LSTFLG
114 IAF8 FECC      CPI      'L'
115 IAF9 C2D719    JNE      P2A
116 IAFF C3F11A    JMP      P2UNLA
117 IB02 C3F11A    JMP      P2UNLA
118
119 *
120 * JUMP ADDRESS TABLE
121
121 IB05 A71C      P2JTBL  DBL     P2C1
122 IB07 E61C      DBL     P2C2
123 IB09 101D      DBL     P2C3
124 IB0B 661D      DBL     P2C4
125 IB0D C61D      DBL     P2C5
126 IB0F 2B1E      DBL     P2C6
127 IB11 971E      DBL     P2C7
128 IB13 A21E      DBL     P2C8
129 IB15 EB1E      DBL     P2C9
130 IB17 1B1F      DBL     P2CA
131 IB19 491F      DBL     P2CB
132 IB1B 8D1F      DBL     P2CC
133 IB1D 3120      DBL     P2CD

```



|     |             |                 |        |                             |
|-----|-------------|-----------------|--------|-----------------------------|
| 134 | 1B1F 3F20   | DBL             | P2CE   |                             |
| 135 | 1B21 6E20   | DBL             | P2CF   |                             |
| 136 | 1B23 AB20   | DBL             | P2C10  |                             |
| 137 | 1B25 D620   | DBL             | P2C11  |                             |
| 138 | 1B27 EB20   | DBL             | P2C12  |                             |
| 139 | 1B29 F520   | DBL             | P2C13  |                             |
| 140 | 1B2B FF20   | DBL             | P2C14  |                             |
| 141 | 1B2D 2921   | DBL             | P2C15  |                             |
| 142 | 1B2F 5321   | DBL             | P2C16  |                             |
| 143 | 1B31 8121   | DBL             | P2C17  |                             |
| 144 | 1B33 C821   | DBL             | P2C18  |                             |
| 145 | 1B35 DB21   | DBL             | P2C19  |                             |
| 146 |             |                 |        |                             |
| 147 |             | *               |        |                             |
| 148 |             | * RES PROCESSOR |        |                             |
| 149 | 1B37 AF     | *               |        |                             |
| 150 | 1B38 32AA22 | P2RES           | ZAR    |                             |
| 151 | 1B3B CD0F15 | STA             | FCFLAG | CLEAR FILL CHAR. FLAG       |
| 152 | 1B3E 4F     | CALL            | FANLYZ |                             |
| 153 | 1B3F CDE51F | MOV             | C,A    | SAVE TERMINATOR             |
| 154 | 1B42 DA821B | CALL            | TCHK1  |                             |
| 155 | 1B45 22EF22 | JC              | P2RESU |                             |
| 156 | 1B48 79     | SHLD            | RSLNTH | SAVE LENGTH                 |
| 157 | 1B49 FE00   | MOV             | A,C    |                             |
| 158 | 1B4B CA7C1B | CPI             | ' '    | WAS THERE A FILL CHARACTER? |
| 159 | 1B4E FE8D   | JEQ             | NOFILL |                             |
| 160 | 1B50 CA7C1B | CPI             | :8D    |                             |
| 161 | 1B53 FEAC   | JEQ             | NOFILL |                             |
| 162 | 1B55 C2821B | CPI             | ' '    | MUST BE COMMA               |
| 163 | 1B58 CD0F15 | JNE             | P2RESU |                             |
| 164 | 1B5B DA9E1B | CALL            | FANLYZ | GET FILL CHARACTER          |
| 165 | 1B5E 78     | JC              | P2RESL | LENGTH ERROR                |
| 166 | 1B5F FE02   | MOV             | A,B    | CHECK TYPE                  |
| 167 | 1B61 FA821B | CPI             | 2      |                             |
| 168 | 1B64 FE20   | JC              | P2RESU |                             |
| 169 | 1B66 DA731B | CPI             | 32     |                             |
| 170 | 1B69 7D     | JAL             | STFILL |                             |
| 171 | 1B6A FE01   | MOV             | A,L    | ASCII, CHECK LENGTH         |
| 172 | 1B6C C2931B | CPI             | 1      | MUST BE 1                   |
| 173 | 1B6F 3AD516 | JNE             | P2RSUF |                             |
| 174 | 1B72 6F     | LDA             | LACHAR | GET FILL CHARACTER          |
| 175 | 1B73 3E01   | MOV             | L,A    |                             |
| 176 | 1B75 32AA22 | STFILL          | MVI    | A,1                         |
| 177 | 1B78 7D     | STA             | FCFLAG | SET FILL FLAG               |
| 178 | 1B79 32AB22 | MOV             | A,L    |                             |
| 179 | 1B7C CD8D23 | STA             | FLCHAR |                             |
| 180 | 1B7F C3F521 | NOFILL          | CALL   | PCHRES                      |
| 181 | 1B82 210000 | JMP             | P2PRNT | PUNCH "RES" RECORD          |
| 182 | 1B85 22EF22 | P2RESU          | LXI    | H,0                         |
| 183 | 1B88 3AAE22 | SHLD            | RSLNTH |                             |
| 184 | 1B8B F601   | LDA             | ERRORS |                             |
| 185 | 1B8D 32AE22 | ORI             | UNDFBT |                             |
| 186 | 1B90 C3F521 | STA             | ERRORS |                             |
| 187 | 1B93 3AAE22 | JMP             | P2PRNT |                             |
| 188 | 1B96 F601   | P2RSUF          | LDA    | ERRORS                      |
| 189 | 1B98 32AE22 | ORI             | UNDFBT |                             |
| 190 | 1B9B C37C1B | STA             | ERRORS |                             |
| 191 | 1B9E 3AAE22 | P2RESL          | JMP    | NOFILL                      |
|     |             |                 | LDA    | ERRORS                      |

```

192 IBA1 F602          ORI   LNTHBT
193 IBA3 32AE22        STA   ERRORS
194 IBA6 C37C1B        JMP   NOFILL
195
196 * ERROR BIT SPECIFICATIONS
197 *
198 UNDFBT EQU 1
199 LNTHBT EQU 2
200 SYMBIT EQU 4
201 MDFBIT EQU 8
202 REBIT  EQU 16
203 OPBIT  EQU 32
204 ADRBIT EQU 64
205 DSPBIT EQU 128
206
207 * "DATA" PROCESSOR
208 *
209 IBA9 21B022 P2DATA LXI   H,0CODE
210 IBAC 22E022 SHLD  P2OPTR
211 IBAF CD0F15 P2DTAB CALL FANLYZ
212 IBB2 DAF21B JC     P2DTA4 LENGTH ERROR
213 IBB5 4F      MOV    C,A
214 IBB6 78      MOV    A,B
215 IBB7 FE02    CPI     2 CHECK TYPE
216 IBB9 FAE71B JC     P2DTA3
217 IBBC FE20    CPI     32
218 IBBE DACB1B JAL     P2DTA2
219 IBC1 7D      MOV    A,L
220 IBC2 FE01    CPI     1 ASCII, MUST BE 1
221 IBC4 C2F21B JNE     P2DTA4
222 IBC7 3AD516 LDA     LACHAR
223 IBCA 6F      MOV    L,A
224 IBCB 3AAF22 P2DTA2 LDA     P2LNTH
225 IBCE 3C      INR     A
226 IBCF 32AF22 STA     P2LNTH
227 IBD2 7D      MOV    A,L
228 IBD3 2AE022 LHLD   P2OPTR
229 IBD6 77      MOV    M,A
230 IBD7 23      INX     H
231 IBD8 22E022 SHLD   P2OPTR
232 IBD8 79      MOV    A,C
233 IBDC FEAC    CPI     ',' LAST ONE?
234 IBDE CAAF1B JEQ     P2DTAB
235 IBE1 CD1223 P2DTND CALL PCHOBJ
236 IBE4 C3F521 JMP     P2PRNT
237 IBE7 3AE22 P2DTA3 LDA     ERRORS
238 IBEA F601    ORI     UNDFBT
239 IBEC 32AE22 P2DC   STA     ERRORS
240 IBEF C3E11B JMP     P2DTND
241 IBF2 3AE22 P2DTA4 LDA     ERRORS
242 IBF5 F602    ORI     LNTHBT
243 IBF7 C3EC1B JMP     P2DC
244
245 * "ASC" PROCESSOR
246 *
247 IBFA 3E01    P2ASC  MVI     A,1
248 IBFC 32AF22 STA     P2LNTH
249 IBFF CD0F15 CALL    FANLYZ

```

|     |             |                   |         |                    |
|-----|-------------|-------------------|---------|--------------------|
| 250 | IC02 4F     | MOV               | C,A     |                    |
| 251 | IC03 78     | MOV               | A,B     |                    |
| 252 | IC04 FE20   | CPI               | 32      |                    |
| 253 | IC06 C22C1C | JNE               | P2ASC4  |                    |
| 254 | IC09 7D     | MOV               | A,L     |                    |
| 255 | IC0A B7     | ORA               | A       |                    |
| 256 | IC0B CA371C | JZ                | P2ASC5  |                    |
| 257 | IC0E FE25   | CPI               | 37      |                    |
| 258 | IC10 D2371C | JGE               | P2ASC5  |                    |
| 259 | IC13 11AB07 | LXI               | D,OPNDF |                    |
| 260 | IC16 13     | INX               | D       | SKIP APOSTROPHE    |
| 261 | IC17 21B022 | LXI               | H,OCODE |                    |
| 262 | IC1A 47     | MOV               | B,A     |                    |
| 263 | IC1B 32AF22 | STA               | P2LNTH  | SAVE LENGTH        |
| 264 | IC1E 1A     | P2ASC2 LDAX       | D       | TRANSMIT STRING    |
| 265 | IC1F 77     | MOV               | M,A     |                    |
| 266 | IC20 13     | INX               | D       |                    |
| 267 | IC21 23     | INX               | H       |                    |
| 268 | IC22 05     | DCR               | B       |                    |
| 269 | IC23 C21E1C | JNZ               | P2ASC2  |                    |
| 270 | IC26 CD1223 | P2ASC3 CALL       | PCHOBJ  |                    |
| 271 | IC29 C3F521 | JMP               | P2PRNT  |                    |
| 272 | IC2C 3AAE22 | P2ASC4 LDA        | ERRORS  |                    |
| 273 | IC2F F601   | ORI               | UNDFBT  |                    |
| 274 | IC31 32AE22 | P2A2 STA          | ERRORS  |                    |
| 275 | IC34 C3261C | JMP               | P2ASC3  |                    |
| 276 | IC37 3AAE22 | P2ASC5 LDA        | ERRORS  |                    |
| 277 | IC3A F602   | ORI               | LNTHBT  |                    |
| 278 | IC3C C3311C | JMP               | P2A2    |                    |
| 279 |             | *                 |         |                    |
| 280 |             | * "REM" PROCESSOR |         |                    |
| 281 |             | *                 |         |                    |
| 282 | IC3F 21A0A0 | P2REM LXI         | H,:A0A0 |                    |
| 283 | IC42 22A507 | SHLD              | MNEM    | BLANK OUT MNEMONIC |
| 284 | IC45 22A707 | SHLD              | MNEM+2  |                    |
| 285 | IC48 C33222 | JMP               | P2PRC   |                    |
| 286 |             | *                 |         |                    |
| 287 |             | * "DBL" PROCESSOR |         |                    |
| 288 |             | *                 |         |                    |
| 289 | IC4B 3E02   | P2DBL MVI         | A,2     |                    |
| 290 | IC4D 32AF22 | STA               | P2LNTH  | SET LENGTH         |
| 291 | IC50 CD0F15 | CALL              | FANLYZ  |                    |
| 292 | IC53 4F     | MOV               | C,A     |                    |
| 293 | IC54 78     | MOV               | A,B     |                    |
| 294 | IC55 FE02   | CPI               | 2       |                    |
| 295 | IC57 FA751C | JC                | P2DBL3  |                    |
| 296 | IC5A FE20   | CPI               | 32      |                    |
| 297 | IC5C C26C1C | JNE               | P2DBL2  |                    |
| 298 | IC5F 7D     | MOV               | A,L     |                    |
| 299 | IC60 FE03   | CPI               | 3       |                    |
| 300 | IC62 D2831C | JGE               | P2DBL5  |                    |
| 301 | IC65 B7     | ORA               | A       |                    |
| 302 | IC66 CA831C | JZ                | P2DBL5  |                    |
| 303 | IC69 2AD516 | LHLD              | LACHAR  |                    |
| 304 | IC6C 22B022 | P2DBL2 SHLD       | OCODE   |                    |
| 305 | IC6F CD1223 | CALL              | PCHOBJ  |                    |
| 306 | IC72 C3F521 | JMP               | P2PRNT  |                    |
| 307 | IC75 3AAE22 | P2DBL3 LDA        | ERRORS  |                    |

```

308 IC78 F601      ORI   UNDFBT
309 IC7A 32AE22    P2DBL4 STA  ERRORS
310 IC7D 210000    LXI   H,0
311 IC80 C36C1C    JMP   P2DBL2
312 IC83 3AAE22    P2DBL5 LDA  ERRORS
313 IC86 F602      ORI   LNTHBT
314 IC88 C37A1C    JMP   P2DBL4
315
316               *
317               * SEARCH SINGLE REGISTER FILE
318               *
318 IC8B 0608      SRF    MVI   B,B
319 IC8D 219F1C    LXI   H,SRFILE
320 IC90 BE        SRF1   CMP   M
321 IC91 CA9B1C    JEQ    SRF2
322 IC94 23        INX    H
323 IC95 05        DCR    B
324 IC96 C2901C    JNZ    SRF1
325 IC99 37        STC                    CARRY ON FOR NO HIT
326 IC9A C9        RET
327 IC9B 05        SRF2   DCR    B
328 IC9C 78        MOV    A,B
329 IC9D B7        RSC                    CLEAR CARRY
330 IC9E C9        RET
331 IC9F C1C0CC08 SRF1   ASC    'AMLHEDCB'
332               *
333               * TYPE 1 PROCESSOR
334               * ARITHMETIC/LOGICAL
335               *
336 ICA7 3E01      P2C1   MVI   A,1
337 ICA9 32AF22    STA    P2LNTH
338 ICAC 1A        LDAX   D
339 ICAD CD8B1C    CALL   SRF      GET REGISTER SPEC
340 ICB0 DADB1C    JC     P2C1C
341 ICB3 F5        PUSH   PSW
342 ICB4 3A8507    LDA    SKEL    CHECK FOR INR/DCR
343 ICB7 FE06      CPI    6
344 ICB9 D2C11C    JGE    PIC2
345 ICBC 78        MOV    A,B      INR OR DCR, MOVE REG.
346 ICB0 87        ADD    A
347 ICBE 87        ADD    A
348 ICBF 87        ADD    A
349 ICC0 47        MOV    B,A
350 ICC1 F1        PIC2   POP    PSW
351 ICC2 13        INX    D
352 ICC3 1A        LDAX   D
353 ICC4 FEA0      CPI    ' '
354 ICC6 CACE1C    JEQ    P2C1A
355 ICC9 FE8D      CPI    8D
356 ICCB C2DB1C    JNE    P2C1C
357 ICCE 3A8507    P2C1A  LDA    SKEL
358 ICD1 B0        ORA    B
359 ICD2 32B022    P2C1AA STA  OCODE
360 ICD5 CD1223    P2C1B  CALL  PCHOBJ
361 ICD8 C3F521    JMP    P2PRNT
362 ICD8 3AAE22    P2C1C  LDA  ERRORS
363 ICDE F610      ORI    REBIT
364 ICE0 32AE22    STA    ERRORS
365 ICE3 C3D51C    JMP    P2C1B

```

```

366
367
368
369
370 ICE6 3E01
371 ICE8 32AF22
372 ICEB 1A
373 ICEC CD8B1C
374 ICEF DADB1C
375 ICF2 87
376 ICF3 87
377 ICF4 87
378 ICF5 4F
379 ICF6 13
380 ICF7 1A
381 ICF8 13
382 ICF9 FEAC
383 ICFB C2DB1C
384 ICFE 1A
385 ICFF CD8B1C
386 ID02 DADB1C
387 ID05 B1
388 ID06 FE36
389 ID08 CADB1C
390 ID0B F640
391 ID0D C3D21C
392
393
394
395
396 ID10 3E02
397 ID12 32AF22
398 ID15 1A
399 ID16 13
400 ID17 CD8B1C
401 ID1A DADB1C
402 ID1D 87
403 ID1E 87
404 ID1F 87
405 ID20 F606
406 ID22 32B022
407 ID25 1A
408 ID26 FEAC
409 ID28 C2DB1C
410 ID2B 13
411 ID2C CD0F15
412 ID2F DA511D
413 ID32 78
414 ID33 FE02
415 ID35 FA5E1D
416 ID38 FE20
417 ID3A DA471D
418 ID3D 7D
419 ID3E FE01
420 ID40 C2511D
421 ID43 3AD516
422 ID46 6F
423 ID47 7D

```

\*  
 \* TYPE 2 PROCESSOR  
 \* MOV  
 \*  
 P2C2    MVI    A,1  
          STA    P2LNTH  
          LDAX   D  
          CALL   SRF  
          JC     P2C1C  
          ADD    A  
          ADD    A  
          ADD    A  
          MOV    C,A  
          INX    D  
          LDAX   D  
          INX    D  
          CPI    ', '  
          JNE    P2C1C      MUST BE COMMA  
          LDAX   D      CHECK SECOND REGISTER  
          CALL   SRF  
          JC     P2C1C  
          ORA    C      MERGE REG. SPECS  
          CPI    066      M.M FORBIDDEN  
          JEQ    P2C1C  
          ORI    40      MERGE OP CODE FOR MOV  
          JMP    P2C1AA  
 \*  
 \* TYPE 3 PROCESSOR  
 \* MVI  
 \*  
 P2C3    MVI    A,2  
          STA    P2LNTH  
          LDAX   D  
          INX    D  
          CALL   SRF  
          JC     P2C1C  
          ADD    A  
          ADD    A  
          ADD    A  
          ORI    6      MERGE WITH MVI SKELETON  
          STA    0CODE  
          LDAX   D  
          CPI    ', '  
          JNE    P2C1C  
          INX    D  
          CALL   FANLYZ    GET IMMEDIATE OPERAND  
          JC     P2C3B  
          MOV    A,B  
          CPI    2      CHECK TYPE  
          JC     P2C3D  
          CPI    32  
          JAL    P2C3A  
          MOV    A,L  
          CPI    1  
          JNE    P2C3B  
          LDA    LACHAR  
          MOV    L,A  
 P2C3A   MOV    A,L



```

424 1D48 32B122          STA  OPCODE+1  SET IMMEDIATE OPERAND
425 1D4B CD1223          CALL PCHOBJ
426 1D4E C3F521          JMP  P2PRNT
427 1D51 3AE22          P2C3B LDA  ERRORS
428 1D54 F602           ORI  LNTHBT
429 1D56 32AE22          P2C3C STA  ERRORS
430 1D59 2E00           MVI  L,0
431 1D5B C3471D          JMP  P2C3A
432 1D5E 3AE22          P2C3D LDA  ERRORS
433 1D61 F601           ORI  UNDFBT
434 1D63 C3561D          JMP  P2C3C
435
436 * TYPE 4 PROCESSOR
437 * MEMORY REFERENCE
438 *
439 1D66 3E03          P2C4  MVI  A,3      SET LENGTH
440 1D68 32AF22          STA  P2LNTH
441 1D6B CD0F15          CALL FANLYZ
442 1D6E CDE51F          CALL TCHK1
443 1D71 DA831D          JC   P2C4B
444 1D74 3A8507          LDA  SKEL
445 1D77 32B022          STA  OPCODE
446 1D7A 22B122          SHLD OPCODE+1
447 1D7D CD1223          P2C4A CALL PCHOBJ
448 1D80 C3F521          JMP  P2PRNT
449 1D83 3AE22          P2C4B LDA  ERRORS
450 1D86 F601           ORI  UNDFBT
451 1D88 32AE22          STA  ERRORS
452 1D8B C37D1D          JMP  P2C4A
453
454 * SEARCH FOR REGISTER PAIR. THIS SEARCH
455 * DEPENDS ON ILLEGAL REGISTER BEING
456 * PREVIOUSLY EXCLUDED.
457 * RETURN WITH CARRY OFF IF R.P. FOUND. RESULT
458 * IN A - 0 FOR B, 1 FOR D, 2 FOR H, 3 FOR SP/PSW
459 *
460 *
461 1D8E 0600          SDRF  MVI  B,0
462 1D90 FEC2          CPI  'B'
463 1D92 CA9E1D          JEQ  SDRFD
464 1D95 04           INR  B
465 1D96 FEC4          CPI  'D'
466 1D98 CA9E1D          JEQ  SDRFD
467 1D9B 04           INR  B
468 1D9C FEC8          CPI  'H'
469 1D9E CA9E1D          JEQ  SDRFD
470 1DA1 04           INR  B
471 1DA2 FED3          CPI  'S'
472 1DA4 C2B11D          JNE  SDRFB
473 1DA7 13           INX  D
474 1DA8 1A           LDAX D
475 1DA9 FED0          CPI  'P'
476 1DAB C2C41D          JNE  SDRFC
477 1DAE 78           SDRFD MOV  A,B
478 1DAF B7           RSC
479 1DB0 C9           RET
480 1DB1 FED0          SDRFB CPI  'P'
481 1DB3 C2C41D          JNE  SDRFC

```

|     |             |                    |       |         |
|-----|-------------|--------------------|-------|---------|
| 482 | IDB6 13     | INX                | D     |         |
| 483 | IDB7 1A     | LDAX               | D     |         |
| 484 | IDB8 FED3   | CPI                | 'S'   |         |
| 485 | IDBA C2C41D | JNE                | SDRFC |         |
| 486 | IDBD 13     | INX                | D     |         |
| 487 | IDBE 1A     | LDAX               | D     |         |
| 488 | IDBF FED7   | CPI                | 'W'   |         |
| 489 | IDC1 CARE1D | JEQ                | SDRFD |         |
| 490 | IDC4 37     | SDRFC              | STC   |         |
| 491 | IDC5 C9     |                    | RET   |         |
| 492 |             | *                  |       |         |
| 493 |             | * TYPE 5 PROCESSOR |       |         |
| 494 |             | * LXI              |       |         |
| 495 |             | *                  |       |         |
| 496 | IDC6 3E03   | P2C5               | MVI   | A,3     |
| 497 | IDC8 32AF22 |                    | STA   | P2LNTH  |
| 498 | IDCB 1A     |                    | LDAX  | D       |
| 499 | IDCC FED0   |                    | CPI   | 'P'     |
| 500 | IDCE CA0B1E |                    | JEQ   | P2C5C   |
| 501 | IDD1 CD8E1D |                    | CALL  | SDRF    |
| 502 | IDD4 DA0B1E |                    | JC    | P2C5C   |
| 503 | IDD7 87     |                    | ADD   | A       |
| 504 | IDD8 87     |                    | ADD   | A       |
| 505 | IDD9 87     |                    | ADD   | A       |
| 506 | IDDA 87     |                    | ADD   | A       |
| 507 | IDDB 3C     |                    | INR   | A       |
| 508 | IDDC 32B022 |                    | STA   | OCODE   |
| 509 | IDDF 13     |                    | INX   | D       |
| 510 | IDE0 1A     |                    | LDAX  | D       |
| 511 | IDE1 FEAC   |                    | CPI   | '/'     |
| 512 | IDE3 C2161E |                    | JNE   | P2C5E   |
| 513 | IDE6 13     |                    | INX   | D       |
| 514 | IDE7 CD0F15 |                    | CALL  | FANLYZ  |
| 515 | IDEA 78     |                    | MOV   | A,B     |
| 516 | IDEB FE02   |                    | CPI   | 2       |
| 517 | IDED FA161E |                    | JC    | P2C5E   |
| 518 | IDF0 FE20   |                    | CPI   | 32      |
| 519 | IDF2 C2021E |                    | JNE   | P2C5AA  |
| 520 | IDF5 7D     |                    | MOV   | A,L     |
| 521 | IDF6 FE03   |                    | CPI   | 3       |
| 522 | IDF8 D2231E |                    | JGE   | P2C5F   |
| 523 | IDFB B7     |                    | ORA   | A       |
| 524 | IDFC CA231E |                    | JZ    | P2C5F   |
| 525 | IDFF 2AD516 |                    | LHLD  | LACHAR  |
| 526 | IE02 22B122 | P2C5AA             | SHLD  | OCODE+1 |
| 527 | IE05 CD1223 | P2C5B              | CALL  | PCHOBJ  |
| 528 | IE08 C3F521 |                    | JMP   | P2PRNT  |
| 529 | IE0B 3AE22  | P2C5C              | LDA   | ERRORS  |
| 530 | IE0E F610   |                    | ORI   | REBIT   |
| 531 | IE10 32AE22 | P2C5D              | STA   | ERRORS  |
| 532 | IE13 C3051E |                    | JMP   | P2C5B   |
| 533 | IE16 FEFE   | P2C5E              | CPI   | -2      |
| 534 | IE18 CA231E |                    | JEQ   | P2C5F   |
| 535 | IE1B 3AE22  |                    | LDA   | ERRORS  |
| 536 | IE1E F601   |                    | ORI   | UNDFBT  |
| 537 | IE20 C3101E |                    | JMP   | P2C5D   |
| 538 | IE23 3AE22  | P2C5F              | LDA   | ERRORS  |
| 539 | IE26 F602   |                    | ORI   | LNTHBT  |

EXCLUDE PSW

SET LOW BIT FOR OPCODE

```

540 IE28 C3101E          JMP    P2C5D
541                      *
542                      * TYPE 6 PROCESSOR
543                      * LDAX,STAX/INX,DCX,DAD/PUSH,POP
544                      *
545 IE2B 3E01             P2C6    MVI    A,1
546 IE2D 32AF22           STA    P2LNTH
547 IE30 3A8507           LDA    SKEL
548 IE33 B7              ORA    A
549 IE34 1F              RAR
550 IE35 DA541E           JC     P2TPP
551 IE38 1A              LDAX   D          LDAX OR STAX
552 IE39 0610            MVI    B,:10
553 IE3B FEC4            CPI    'D'
554 IE3D CA471E           JEQ    P2C6A
555 IE40 0600            MVI    B,0
556 IE42 FEC2            CPI    'B'
557 IE44 C28B1E           JNE    P2C6D
558 IE47 3A8507           P2C6A   LDA    SKEL
559 IE4A B0              P2C6A1  ORA    B
560 IE4B 32B022           P2C6B   STA    0CODE
561 IE4E CD1223           CALL   PCH0BJ
562 IE51 C3F521           JMP    P2PRNT
563 IE54 3A8507           P2TPP   LDA    SKEL
564 IE57 FEC1            CPI    :C1
565 IE59 CA821E           JEQ    P2PP
566 IE5C FEC5            CPI    :C5
567 IE5E CA821E           JEQ    P2PP          CHECK FOR PUSH OR POP
568 IE61 1A              LDAX   D          MUST BE INX, DCX OR DAD
569 IE62 FED0            CPI    'P'          ELIMINATE PSW
570 IE64 CA8B1E           JEQ    P2C6D
571 IE67 CD8E1D           P2PP1   CALL   SDRF
572 IE6A DA8B1E           JC     P2C6D
573 IE6D 13              INX    D
574 IE6E 1A              LDAX   D
575 IE6F FEA0            CPI    ' '
576 IE71 CA791E           JEQ    P2C6C
577 IE74 FE8D            CPI    :8D
578 IE76 C28B1E           JNE    P2C6D
579 IE79 78              P2C6C   MOV    A,B
580 IE7A 87              ADD    A
581 IE7B 87              ADD    A
582 IE7C 87              ADD    A
583 IE7D 87              ADD    A
584 IE7E 47              MOV    B,A
585 IE7F C3471E           JMP    P2C6A
586 IE82 1A              P2PP    LDAX   D          PUSH OR POP
587 IE83 FED3            CPI    'S'          ELIMINATE STACK POINTER
588 IE85 CA8B1E           JEQ    P2C6D
589 IE88 C3671E           JMP    P2PP1
590 IE8B 3A8E22           P2C6D   LDA    ERRORS
591 IE8E F610            ORI    REBIT
592 IE90 32AE22           STA    ERRORS
593 IE93 AF              ZAR
594 IE94 C34B1E           JMP    P2C6B
595                      *
596                      * TYPE 7 PROCESSOR
597                      *

```

```

598 IE97 3E01 P2C7 MVI A.1
599 IE99 32AF22 STA P2LNTH
600 IE9C 3A8507 LDA SKEL
601 IE9F C34B1E JMP P2C6B
602
603 * TYPE 8 PROCESSOR
604 *
605 IEA2 3E02 P2C8 MVI A.2
606 IEA4 32AF22 STA P2LNTH
607 IEA7 3A8507 LDA SKEL
608 IEAA 32B022 STA OC0DE
609 IEAD CD0F15 CALL FANLYZ
610 IEB0 DAE31E JC P2C8F
611 IEB3 78 MOV A.B
612 IEB4 FE02 CPI 2
613 IEB6 FAD71E JC P2C8D
614 IEB9 FE20 CPI 32
615 IEBB CACB1E JEQ P2C8B
616 IEBE D2D71E JGE P2C8D
617 IEC1 7D MOV A.L
618 IEC2 32B122 P2C8A STA OC0DE+1
619 IEC5 CD1223 CALL PCH0BJ
620 IEC8 C3F521 JMP P2PRNT
621 IECB 7D P2C8B MOV A.L
622 IECC FE01 CPI 1
623 IECE C2E31E JNE P2C8F
624 IED1 3AD516 LDA LACHAR
625 IED4 C3C21E JMP P2C8A
626 IED7 3AE22 P2C8D LDA ERRORS
627 IEDA F601 ORI UNDFBT
628 IEDC 32AE22 P2C8E STA ERRORS
629 IEDF AF ZAR
630 IEE0 C3C21E JMP P2C8A
631 IEE3 3AE22 P2C8F LDA ERRORS
632 IEE6 F602 ORI LNTHBT
633 IEE8 C3DC1E JMP P2C8E
634
635 * TYPE 9 PROCESSOR
636 *
637 IEEB 3E02 P2C9 MVI A.2
638 IEE0 32AF22 STA P2LNTH SET LENGTH
639 IEF0 3A8507 LDA SKEL
640 IEF3 FE00 CPI 0
641 IEF5 CA091F JEQ P2LRA
642 IEF8 FE01 CPI 1
643 IEFA CA0F1F JEQ P2TCA
644 IEFD 2A151F LHLD STPC0D
645 IF00 22B022 P2C9A SHLD OC0DE
646 IF03 CD1223 P2C9B CALL PCH0BJ
647 IF06 C3F521 JMP P2PRNT
648 IF09 2A171F P2LRA LHLD LRAC0D
649 IF0C C3001F JMP P2C9A
650 IF0F 2A191F P2TCA LHLD TCAC0D
651 IF12 C3001F JMP P2C9A
652 IF15 FB STPC0D EI
653 IF16 76 HLT
654 IF17 B7 LRAC0D ORA A
655 IF18 1F RAR

```

```

656 IF19 2F          TCACOD CMA
657 IF1A 3C          INR  A
658
659                  *
660                  * TYPE A PROCESSOR
661                  *
661 IF1B 3E01         P2CA  MVI  A,1
662 IF1D 32AF22       STA  P2LNTH
663 IF20 CD0F15       CALL FANLYZ
664 IF23 CDE51F       CALL TCHK1
665 IF26 DA3E1F       JC   P2CAE
666 IF29 7D          MOV  A,L
667 IF2A B7          ORA  A
668 IF2B FA3E1F       JM   P2CAE
669 IF2E FE08         CPI  8
670 IF30 D23E1F       JGE  P2CAE
671 IF33 87          ADD  A
672 IF34 87          ADD  A
673 IF35 87          ADD  A
674 IF36 F6C7        ORI  C7
675 IF38 32B022       P2CAB STA  OPCODE
676 IF3B C3031F       JMP  P2C9B
677 IF3E 3A8E22       P2CAE LDA  ERRORS
678 IF41 F601         ORI  UNDFBT
679 IF43 328E22       STA  ERRORS
680 IF46 C3031F       JMP  P2C9B
681
682                  *
683                  *
684                  *
685                  * Z80 INSTRUCTION TYPES B THROUGH 19
686                  *
687                  *
688                  *
689                  *
690                  * CLASS B
691                  *
692 IF49 3E03         P2CB  MVI  A,3
693 IF4B 32AF22       STA  P2LNTH
694 IF4E 3A8807       LDA  MNEM+3
695 IF51 FE08         CPI  /
696 IF53 C2591F       JNE  P2CB1
697 IF56 3A8707       LDA  MNEM+2
698 IF59 CDDA1F       P2CB1 CALL DDFDXY
699 IF5C 32B022       STA  OPCODE
700 IF5F 3A8507       LDA  SKEL
701 IF62 32B122       STA  OPCODE+1
702 IF65 CD0F15       CALL FANLYZ
703 IF68 CDE51F       CALL TCHK1
704 IF6B DA1120       JC   P2ZUER
705 IF6E CDF41F       CALL ZL7CHK
706 IF71 DA2920       JC   P2ZDER
707 IF74 22B222       SHLD OPCODE+2
708
709                  *
710                  * FOLLOWING FLAGS Z80 ONLY INSTRUCTIONS WITH
711                  * A DOLLAR SIGN ($) TWO COLUMNS BEFORE THE
712                  * LABEL IN THE ASSEMBLY LISTING.
713                  *
713 IF77 3EA4         FLGZ80 MVI  A,'$'

```



```

714 IF79 329B07      STA LABEL-2
715 IF7C 3A8C1F      LDA IL8080
716 IF7F B7          ORA A
717 IF80 CAD51C      JZ P2C1B
718 IF83 218907      LXI H,HEX
719 IF86 CD9901      CALL PRT
720 IF89 C3D51C      JMP P2C1B
721 IF8C 00          IL8080 ***
722                  *
723                  * CLASS C
724                  *
725 IF8D 3E04          P2CC MVI A,4
726 IF8F 32AF22      STA P2LNTH
727 IF92 3AA707      LDA MNEM+2
728 IF95 CDDA1F      CALL DDFDXY
729 IF98 32B022      STA OCODE
730 IF9B 3E36          MVI A,:36
731 IF9D 32B122      STA OCODE+1
732 IFA0 CD0F15      CALL FANLY2
733 IFA3 FEAC          CPI ','
734 IFA5 C21120      JNE P2ZUER
735 IFA8 CDE51F      CALL TCHK1
736 IFAB DA1120      JC P2ZUER
737 IFAE CDF41F      CALL ZL7CHK
738 IFB1 DA2920      JC P2ZDER
739 IFB4 22B222      SHLD OCODE+2
740 IFB7 CD0F15      CALL FANLY2
741 IFBA DA0620      JC P2ZLER
742 IFBD 78          MOV A,B
743 IFBE FE02          CPI 2
744 IFC0 FA1120      J< P2ZUER
745 IFC3 FE20          CPI 32
746 IFC5 DAD41F      JAL P2CC2
747 IFC8 C21120      JNE P2ZUER
748 IFCB 7D          MOV A,L
749 IFCC FE01          CPI 1
750 IFCE C20620      JNE P2ZLER
751 IFD1 2AD516      LHLD LACHAR
752 IFD4 22B322      P2CC2 SHLD OCODE+3
753 IFD7 C3771F      JMP FLG280
754                  *
755                  * DETERMINE :DD OR :FD PREFIX FROM
756                  * PRESENCE OF "X" OR "Y"
757                  *
758 IFDA 06DD          DDFDXY MVI B,:DD
759 IFDC FED8          CPI 'X'
760 IFDE CAE31F      JEQ DDFDZ2
761 IFE1 06FD          MVI B,:FD
762 IFE3 78          DDFDZ2 MOV A,B
763 IFE4 C9          RET
764                  *
765                  * CHECK OPERAND TYPE LIMITS
766                  *
767 IFE5 78          TCHK1 MOV A,B
768 IFE6 FE02          CPI 2
769 IFEB FAF21F      J< TCHK2
770 IFEB FE20          CPI 32
771 IFED D2F21F      JGE TCHK2

```

MUST END IN COMMA

```

772 IFF0 B7          RSC
773 IFF1 C9          RET
774 IFF2 37          TCHK2  STC
775 IFF3 C9          RET
776
777 *
778 * CHECK FANLY2 RESULT FOR VALID SINGLE
779 * PRECISION SIGNED NUMBER. RETURN WITH
780 * CARRY ON IF GREATER THAN SINGLE
781 * PRECISION (7 BITS + SIGN) IN H&L.
782 IFF4 7C          ZL7CHK  MOV  A,H
783 IFF5 B7          ORA  A
784 IFF6 CA0020       JZ  ZL7C2
785 IFF9 FEFF        CPI  :FF
786 IFFB CA0020       JEQ  ZL7C2
787 IFFE 37          ZL7C1  STC
788 IFFF C9          RET
789 2000 AD          ZL7C2  L  L
790 2001 FAFE1F       JM  ZL7C1
791 2004 B7          RSC
792 2005 C9          RET
793
794 *
795 * Z80 ERROR HANDLERS
796 2006 3AAE22       P2ZLER  LDA  ERRORS
797 2009 F602         ORI  LNTHBT
798 200B 32AE22       P2ZL2  STA  ERRORS
799 200E C3771F       JMP  FLGZ80
800 2011 3AAE22       P2ZUER  LDA  ERRORS
801 2014 F601         ORI  UNDFBT
802 2016 C30B20       JMP  P2ZL2
803 2019 3AAE22       P2ZRER  LDA  ERRORS
804 201C F610         ORI  REBIT
805 201E C30B20       JMP  P2ZL2
806 2021 3AAE22       P2ZARER LDA  ERRORS
807 2024 F640         ORI  ADRBIT
808 2026 C30B20       JMP  P2ZL2
809 2029 3AAE22       P2ZDER  LDA  ERRORS
810 202C F680         ORI  DSPBIT
811 202E C30B20       JMP  P2ZL2
812
813 *
814 * CLASS D
815
816 2031 3E01         P2CD  MVI  A,1
817 2033 32AF22       STA  P2LNTH
818 2036 3A8507       LDA  SKEL
819 2039 32B022       STA  OCODE
820 203C C3771F       JMP  FLGZ80
821
822 *
823 * CLASS E
824 203F 3E02         P2CE  MVI  A,2
825 2041 32AF22       STA  P2LNTH
826 2044 3EED        MVI  A,:ED
827 2046 32B022       STA  OCODE
828 2049 3AA607       LDA  MNEM+1
829 204C 0E0A        MVI  C,:A

```

```

830 204E FEC1          CPI    'A'
831 2050 CA5520        JEQ    P2CE2
832 2053 0E02          MVI    C,2
833 2055 1A           P2CE2 LDAX    D
834 2056 FED0          CPI    'P'
835 2058 CA1920        JEQ    P2ZRER
836 205B CD8E1D        CALL   SDRF
837 205E DA1920        JC     P2ZRER
838 2061 C604          ADI    4
839 2063 87           ADD    A
840 2064 87           ADD    A
841 2065 87           ADD    A
842 2066 87           ADD    A
843 2067 B1           ORA    C
844 2068 32B122        STA    OCODE+1
845 206B C3771F        JMP    FLGZ80
846
847
848
* CLASS F - LXI X/Y - FED BY 8080 CLASS 5
*
849 206E 3E04          P2CF  MVI    A,4
850 2070 32AF22        STA    P2LNTH
851 2073 1A           LDAX    D
852 2074 CDDA1F        CALL   DDFDXY
853 2077 32B022        STA    OCODE
854 207A 3E21          MVI    A,:21
855 207C 32B122        STA    OCODE+1
856 207F 13           INX    D
857 2080 1A           LDAX    D
858 2081 FEAC          CPI    ','      MUST BE COMMA
859 2083 C21120        JNE    P2ZUER
860 2086 13           INX    D
861 2087 CD0F15        CALL   FANLYZ
862 208A 78           MOV    A,B
863 208B FE02          CPI    2
864 208D FA1120        JC     P2ZUER
865 2090 FE20          CPI    32
866 2092 DAA520        JAL    P2CF3
867 2095 C21120        JNE    P2ZUER
868 2098 7D           P2CF2 MOV    A,L
869 2099 FE03          CPI    3
870 209B D20620        JGE    P2ZLER
871 209E B7           ORA    A
872 209F CA0620        JZ     P2ZLER
873 20A2 2AD516        LHALD  LACHAR
874 20A5 22B222        P2CF3 SHLD   OCODE+2
875 20A8 C3771F        JMP    FLGZ80
876
877
878
* CLASS 10
*
879 20AB 3E04          P2C10 MVI    A,4
880 20AD 32AF22        STA    P2LNTH
881 20B0 06ED          MVI    B,:ED
882 20B2 3AA807        LDA    MNEM+3
883 20B5 FE00          CPI    ' '
884 20B7 C2C020        JNE    P2C10B
885 20BA 3AA607        LDA    MNEM+1
886 20BD CDDA1F        CALL   DDFDXY
887 20C0 78           P2C10B MOV    A,B

```

```

888 20C1 32B022      STA  OCODE
889 20C4 3A8507      LDA  SKEL
890 20C7 32B122      STA  OCODE+1
891 20CA CD0F15      CALL FANLYZ
892 20CD CDE51F      CALL TCHK1
893 20D0 DA1120      JC   P2ZUER
894 20D3 C3A520      JMP  P2CF3
895
896 * CLASS 11
897 *
898 20D6 3E02      P2C11 MVI  A,2
899 20D8 32AF22      STA  P2LNTH
900 20DB 1A        LDAX  D
901 20DC CDDA1F      CALL DDFDXY
902 20DF 32B022      P2C11A STA  OCODE
903 20E2 3A8507      LDA  SKEL
904 20E5 32B122      STA  OCODE+1
905 20E8 C3771F      JMP  FLG280
906
907 * CLASS 12
908 *
909 20EB 3E02      P2C12 MVI  A,2
910 20ED 32AF22      STA  P2LNTH
911 20F0 3EED      MVI  A,:ED
912 20F2 C3DF20      JMP  P2C11A
913
914 * CLASS 13
915 *
916 20F5 3E02      P2C13 MVI  A,2
917 20F7 32AF22      STA  P2LNTH
918 20FA 3ECB      MVI  A,:CB
919 20FC C3DF20      JMP  P2C11A
920
921 * CLASS 14
922 *
923 20FF 3E02      P2C14 MVI  A,2
924 2101 32AF22      STA  P2LNTH
925 2104 3A8507      LDA  SKEL
926 2107 32B022      STA  OCODE
927 210A CD0F15      CALL FANLYZ
928 210D CDE51F      CALL TCHK1
929 2110 DA1120      JC   P2ZUER
930 2113 EB        XCHG
931 2114 2AB818      LHLD  ILC
932 2117 23        INX  H
933 2118 23        INX  H
934 2119 CD3304      CALL NEGHL
935 211C 19        DAD  D
936 211D CDF41F      CALL ZL7CHK
937 2120 DA2120      JC   P2ZUER
938 2123 22B122      SHLD  OCODE+1
939 2126 C3771F      JMP  FLG280
940
941 * CLASS 15
942 *
943 2129 3E02      P2C15 MVI  A,2
944 212B 32AF22      STA  P2LNTH
945 212E 3ECB      MVI  A,:CB

```

```

946 2130 32B022      STA  OPCODE
947 2133 CD0F15      CALL FANLYZ
948 2136 DA0620      JC   P2ZLER
949 2139 CDE51F      CALL TCHK1
950 213C DA1120      JC   P2ZUER
951 213F 7D          MOV  A,L
952 2140 FE08        CPI  8
953 2142 D20620      JGE  P2ZLER
954 2145 87          ADD  A
955 2146 87          ADD  A
956 2147 87          ADD  A
957 2148 47          MOV  B,A
958 2149 3A8507      LDA  SKEL
959 214C B0          ORA  B
960 214D 32B122      STA  OPCODE+1
961 2150 C3771F      JMP  FLGZ80
962
963
964
965 2153 3E04        MVI  A,4
966 2155 32AF22      STA  P2LNTH
967 2158 3A8807      LDA  MNEM+3
968 215B CDDA1F      CALL DDFDXY
969 215E 32B022      STA  OPCODE
970 2161 3ECB        MVI  A,:CB
971 2163 32B122      STA  OPCODE+1
972 2166 CD0F15      CALL FANLYZ
973 2169 CDE51F      CALL TCHK1
974 216C DA1120      JC   P2ZUER
975 216F CDF41F      CALL ZL7CHK
976 2172 DA2920      JC   P2ZDER
977 2175 22B222      SHLD OPCODE+2
978 2178 3A8507      LDA  SKEL
979 217B 32B322      STA  OPCODE+3
980 217E C3771F      JMP  FLGZ80
981
982
983
984 2181 3E04        MVI  A,4
985 2183 32AF22      STA  P2LNTH
986 2186 3A8807      LDA  MNEM+3
987 2189 CDDA1F      CALL DDFDXY
988 218C 32B022      STA  OPCODE
989 218F 3ECB        MVI  A,:CB
990 2191 32B122      STA  OPCODE+1
991 2194 CD0F15      CALL FANLYZ
992 2197 FERC        CPI  ' '
993 2199 C21120      JNE  P2ZUER
994 219C CDE51F      CALL TCHK1
995 219F DA1120      JC   P2ZUER
996 21A2 CDF41F      CALL ZL7CHK
997 21A5 DA2920      JC   P2ZDER
998 21A8 22B222      SHLD OPCODE+2
999 21AB CD0F15      CALL FANLYZ
000 21AE 78          MOV  A,B
001 21AF FE02        CPI  2
002 21B1 FA1120      JC   P2ZUER
003 21B4 7D          MOV  A,L

```



```

004 21B5 FE08          CPI      8
005 21B7 D20620        JGE      P2ZLER
006 21BA 87            ADD      A
007 21BB 87            ADD      A
008 21BC 87            ADD      A
009 21BD 47            MOV      B,A
010 21BE 3A8507        LDA      SKEL
011 21C1 B0            ORA      B
012 21C2 32B322        STA      OCODE+3
013 21C5 C3771F        JMP      FLGZ80
014
015 *
016 * CLASS 18
017 21C8 3E02          P2C18    MVI      A,2
018 21CA 32AF22        STA      P2LNTH
019 21CD 3EDD          MVI      A,:DD
020 21CF 32B022        P2C18A   STA      OCODE
021 21D2 3A8507        LDA      SKEL
022 21D5 32B122        STA      OCODE+1
023 21D8 C3771F        JMP      FLGZ80
024
025 *
026 * CLASS 19
027 21DB 3E02          P2C19    MVI      A,2
028 21DD 32AF22        STA      P2LNTH
029 21E0 3EFD          MVI      A,:FD
030 21E2 C3CF21        JMP      P2C18A
031
032 *
033 *
034 * BAD MNEMONIC PROCESSOR
035 *
036 21E5 3E04          P2BDOP    MVI      A,4
037 21E7 32AF22        STA      P2LNTH
038 21EA 3AAE22        LDA      ERRORS
039 21ED F620          ORI      OPBIT
040 21EF 32AE22        STA      ERRORS
041 21F2 C3031F        JMP      P2C9B
042
043 *
044 * PRINT MACHINERY
045 21F5 3AB918        P2PRNT    LDA      ILC+1      CONVERT LOCATION
046 21F8 CD8513        CALL     CNV2HX
047 21FB 328D07        STA      HEX+4
048 21FE 78            MOV      A,B
049 21FF 328E07        STA      HEX+5
050 2202 3AB818        LDA      ILC
051 2205 CD8513        CALL     CNV2HX
052 2208 328F07        STA      HEX+6
053 220B 78            MOV      A,B
054 220C 329007        STA      HEX+7
055 220F 3AAF22        LDA      P2LNTH
056 2212 B7            ORA      A      CHECK FOR ZERO
057 2213 CA3222        JZ       P2PRC
058 2216 FE05          CPI      5
059 2218 DA1D22        JAL      P2PRA      SET MAXIMUM 4 BYTES
060 221B 3E04          MVI      A,4
061 221D 4F            P2PRA     MOV      C,A      COUNT TO C

```

|                 |        |      |          |                          |
|-----------------|--------|------|----------|--------------------------|
| 062 221E 21B022 |        | LXI  | H, 0CODE |                          |
| 063 2221 119207 |        | LXI  | D, HEX+9 |                          |
| 064 2224 7E     | P2PRB  | MOV  | A, M     | CONVERT LOOP             |
| 065 2225 23     |        | INX  | H        |                          |
| 066 2226 CD8513 |        | CALL | CONV2HX  |                          |
| 067 2229 12     |        | STAX | D        |                          |
| 068 222A 13     |        | INX  | D        |                          |
| 069 222B 78     |        | MOV  | A, B     |                          |
| 070 222C 12     |        | STAX | D        |                          |
| 071 222D 13     |        | INX  | D        |                          |
| 072 222E 00     |        | DCR  | C        |                          |
| 073 222F C22422 |        | JNZ  | P2PRB    |                          |
| 074 2232 3AAE22 | P2PRC  | LDA  | ERRORS   |                          |
| 075 2235 B7     |        | ORA  | A        |                          |
| 076 2236 CA5C22 |        | JZ   | P2PRCA   |                          |
| 077 2239 0E08   |        | MVI  | C, 8     | CONVERT ERROR LINE       |
| 078 223B 21D822 |        | LXI  | H, ERCDZ |                          |
| 079 223E 11E622 |        | LXI  | D, ECA   |                          |
| 080 2241 46     | P2PRG  | MOV  | B, M     |                          |
| 081 2242 1F     |        | RAR  |          |                          |
| 082 2243 D24B22 |        | JNC  | P2PRH    |                          |
| 083 2246 F5     |        | PUSH | PSW      |                          |
| 084 2247 78     |        | MOV  | A, B     |                          |
| 085 2248 12     |        | STAX | D        |                          |
| 086 2249 F1     |        | POP  | PSW      |                          |
| 087 224A 13     |        | INX  | D        |                          |
| 088 224B 23     | P2PRH  | INX  | H        |                          |
| 089 224C 00     |        | DCR  | C        |                          |
| 090 224D C24122 |        | JNZ  | P2PRG    |                          |
| 091 2250 218907 |        | LXI  | H, HEX   |                          |
| 092 2253 CD9901 |        | CALL | PRT      |                          |
| 093 2256 21E222 |        | LXI  | H, ERLNE |                          |
| 094 2259 CD9901 |        | CALL | PRT      |                          |
| 095 225C 3AA013 | P2PRCA | LDA  | PRFLAG   |                          |
| 096 225F B7     |        | ORA  | A        |                          |
| 097 2260 CA7622 |        | JZ   | P2PRF    |                          |
| 098 2263 218907 | P2PRD  | LXI  | H, HEX   |                          |
| 099 2266 CD6C14 |        | CALL | LPNT     | PRINT LINE               |
| 100 2269 3AAE22 |        | LDA  | ERRORS   | CHECK FOR ERROR FLAGS    |
| 101 226C B7     |        | ORA  | A        |                          |
| 102 226D CA7622 |        | JZ   | P2PRF    |                          |
| 103 2270 21E222 |        | LXI  | H, ERLNE |                          |
| 104 2273 CD6C14 |        | CALL | LPNT     |                          |
| 105 2276 3A8407 | P2PRF  | LDA  | CLASS    | CHECK FOR "END"          |
| 106 2279 FE80   |        | CPI  | :80      |                          |
| 107 227B C28622 |        | JNE  | P2PRI    |                          |
| 108 227E 3A8507 |        | LDA  | SKEL     |                          |
| 109 2281 FE01   |        | CPI  | 1        |                          |
| 110 2283 CA9B22 |        | JEQ  | PTRLR    |                          |
| 111 2286 3AAF22 | P2PRI  | LDA  | P2LNTH   |                          |
| 112 2289 4F     |        | MOV  | C, A     |                          |
| 113 228A 0600   |        | MVI  | B, 0     |                          |
| 114 228C 2AB818 |        | LHLD | ILC      | UPDATE LOCATION COUNTER  |
| 115 228F 09     |        | DAD  | B        |                          |
| 116 2290 EB     |        | XCHG |          |                          |
| 117 2291 2AEF22 |        | LHLD | RSLNTH   |                          |
| 118 2294 19     |        | DAD  | D        |                          |
| 119 2295 22B818 |        | SHLD | ILC      | RESTORE LOCATION COUNTER |

```

120 2298 C3D719      JMP P2A      AND GET NEXT LINE
121 229B CD3C23      PTRLR  CALL DPOBJR
122 229E 3E46        MVI A,70
123 22A0 CDF122      CALL PCHBLT
124 22A3 C9          RET
125 22A4 310001      RTMON  LXI SP,MON
126 22A7 C31901      JMP MON2
127                  *
128 22AA 00          FCFLAG ***
129 22AB 00          FLCHAR ***
130 22AC 0000        P2LNO  DBL 0
131 22AE 00          ERRORS ***
132 22AF 00          P2LNTH ***
133 22B0             OCODE  RES 40
134 22D8 D5CCD3CD    ERCDZ  ASC 'ULSMROAD'
135 22E0 0000        P2OPTR DBL 0
136 22E2 A0A0A0A0    ERLNE  ASC '
137 22E6             ECA     RES 8
138 22EE 8D          DATA  :8D
139 22EF 0000        RSLNTH DBL 0
140                  *
141                  *****
142                  *
143                  * OBJECT OUTPUT DRIVERS
144                  *
145                  *
146                  * PUNCH LEADER/TRAILER
147                  *
148 22F1 47          PCHBLT MOV B,A
149 22F2 3AA113      LDA OBFLAG
150 22F5 B7           ORA A
151 22F6 CA0123      JZ PCHBL3
152 22F9 AF          ZAR
153 22FA CD7A01      PCHBL2 CALL PCHCHR
154 22FD 05          DCR B
155 22FE C2FA22      JNZ PCHBL2
156 2301 210000      PCHBL3 LXI H,0
157 2304 22E023      SHLD RCDORG  CLEAR RECORD ORIGIN
158 2307 AF          ZAR
159 2308 32E223      STA RCDISZ
160 230B 21E523      LXI H,RCDBLK
161 230E 22E323      SHLD RCDPTR
162 2311 C9          RET
163                  *
164                  * ENTER OBJECT INTO PUNCH BUFFER
165                  *
166 2312 3AAF22      PCHOBJ LDA P2LNTH
167 2315 B7           ORA A
168 2316 C8          RZ
169 2317 47          MOV B,A
170 2318 3AE223      LDA RCDISZ
171 231B 80          ADD B
172 231C FE81        CPI :81  OVER 128?
173 231E DA2723      JAL PCHOBJ2
174 2321 CD3C23      CALL DPOBJR  YES, DUMP EXISTING RECORD
175 2324 C31223      JMP PCHOBJ  AND THEN DO IT
176 2327 32E223      PCHOBJ2 STA RCDISZ
177 232A 11B022      LXI D,OCODE

```

|                 |                           |          |                       |
|-----------------|---------------------------|----------|-----------------------|
| 178 232D 2AE323 | LHLD                      | RCDPTR   |                       |
| 179 2330 1A     | PCHOB3 LDAX               | D        |                       |
| 180 2331 77     | MOV                       | M,A      | TRANSMIT OBJECT CODE  |
| 181 2332 13     | INX                       | D        |                       |
| 182 2333 23     | INX                       | H        |                       |
| 183 2334 05     | DCR                       | B        |                       |
| 184 2335 C23023 | JNZ                       | PCHOB3   |                       |
| 185 2338 22E323 | SHLD                      | RCDPTR   |                       |
| 186 233B C9     | RET                       |          |                       |
| 187             | *                         |          |                       |
| 188             | * PUNCH OUT OBJECT RECORD |          |                       |
| 189             | *                         |          |                       |
| 190 233C 3AA113 | DPOBJR LDA                | OBFLAG   |                       |
| 191 233F B7     | ORA                       | A        |                       |
| 192 2340 CA7C23 | JZ                        | DPOBJ3   |                       |
| 193 2343 3AE223 | LDA                       | RCD512   |                       |
| 194 2346 B7     | ORA                       | A        |                       |
| 195 2347 CA7C23 | JZ                        | DPOBJ3   |                       |
| 196 234A 47     | MOV                       | B,A      |                       |
| 197 234B AF     | ZAR                       |          |                       |
| 198 234C CD7A01 | CALL                      | PCHCHR   |                       |
| 199 234F 3D     | DCR                       | A        |                       |
| 200 2350 1600   | MVI                       | D,0      | CLEAR CHECKSUM        |
| 201 2352 CDD823 | CALL                      | PCHFRM   | PUNCH RECORD MARK     |
| 202 2355 2AE023 | LHLD                      | RCDORG   |                       |
| 203 2358 7D     | MOV                       | A,L      |                       |
| 204 2359 CDD823 | CALL                      | PCHFRM   |                       |
| 205 235C 7C     | MOV                       | A,H      |                       |
| 206 235D CDD823 | CALL                      | PCHFRM   | PUNCH ORIGIN          |
| 207 2360 AF     | ZAR                       |          |                       |
| 208 2361 CDD823 | CALL                      | PCHFRM   | PUNCH RECORD TYPE     |
| 209 2364 78     | MOV                       | A,B      |                       |
| 210 2365 CDD823 | CALL                      | PCHFRM   | PUNCH RECORD LENGTH   |
| 211 2368 21E523 | LXI                       | H,RCDBLK |                       |
| 212 236B 7E     | DPOBJ2 MOV                | A,M      |                       |
| 213 236C 23     | INX                       | H        |                       |
| 214 236D CDD823 | CALL                      | PCHFRM   |                       |
| 215 2370 05     | DCR                       | B        |                       |
| 216 2371 C26B23 | JNZ                       | DPOBJ2   |                       |
| 217 2374 7A     | MOV                       | A,D      |                       |
| 218 2375 CD7A01 | CALL                      | PCHCHR   | PUNCH CHECKSUM        |
| 219 2378 AF     | ZAR                       |          |                       |
| 220 2379 CD7A01 | CALL                      | PCHCHR   |                       |
| 221 237C 2AB818 | DPOBJ3 LHLD               | ILC      | SET NEW RECORD ORIGIN |
| 222 237F 22E023 | SHLD                      | RCDORG   |                       |
| 223 2382 AF     | ZAR                       |          |                       |
| 224 2383 32E223 | STA                       | RCD512   |                       |
| 225 2386 21E523 | LXI                       | H,RCDBLK | RESET ADDRESS POINTER |
| 226 2389 22E323 | SHLD                      | RCDPTR   |                       |
| 227 238C C9     | RET                       |          |                       |
| 228             | *                         |          |                       |
| 229             | * PUNCH "RES" TYPE RECORD |          |                       |
| 230             | *                         |          |                       |
| 231 238D CD3C23 | PCHRES CALL               | DPOBJR   | CLEAR PREVIOUS RECORD |
| 232 2390 3AA113 | LDA                       | OBFLAG   |                       |
| 233 2393 B7     | ORA                       | A        |                       |
| 234 2394 CACC23 | JZ                        | PCHRS3   |                       |
| 235 2397 2100FF | LXI                       | H, :FF00 |                       |

```

236 239A 22E523      SHLD  RCDBLK
237 239D 2AE023      LHL  RCDORG
238 23A0 22E723      SHLD  RCDBLK+2  SET UP PUNCH RECORD
239 23A3 3E01        MVI   A,1      RECORD TYPE
240 23A5 32E923      STA   RCDBLK+4
241 23A8 2AAA22      LHL  FCFLAG
242 23AB 22EA23      SHLD  RCDBLK+5
243 23AE 2AEF22      LHL  RSLNTH
244 23B1 22EC23      SHLD  RCDBLK+7
245 23B4 0609        MVI   B,9
246 23B6 21E523      LXI   H,RCDBLK
247 23B9 1600        MVI   D,0      CLEAR CHECKSUM
248 23BB 7E          PCHRS2 MOV   A,M
249 23BC CDD823      CALL  PCHFRM
250 23BF 23          INX   H
251 23C0 05          DCR   B
252 23C1 C2BB23      JNZ   PCHRS2
253 23C4 7A          MOV   A,D
254 23C5 CD7A01      CALL  PCHCHR  PUNCH CHECKSUM
255 23C8 AF          ZAR
256 23C9 CD7A01      CALL  PCHCHR
257 23CC 2AEF22      PCHRS3 LHL  RSLNTH
258 23CF EB          XCHG
259 23D0 2AB818      LHL  ILC      UPDATE RECORD ORIGIN
260 23D3 19          DAD   D
261 23D4 22E023      SHLD  RCDORG
262 23D7 C9          RET
263
264      *
265      * PUNCH FRAME AND KEEP CHECKSUM
266      *
266 23D8 F5          PCHFRM PUSH  PSW
267 23D9 82          ADD   D
268 23DA 57          MOV   D,A
269 23DB F1          POP   PSW
270 23DC CD7A01      CALL  PCHCHR
271 23DF C9          RET
272
273 23E0 0000        *
274 23E2 00          RCDORG DBL   0
275 23E3 0000        RCDISZ ****
276 23E5            RCDPTR DBL   0
277 2449            RCDBLK RES   100
278            RES   28
279 2465            EORSM EQU   RCDBLK+128
                END

```

\*\*\*\*\*  
\* S Y M B O L T A B L E \*  
\*\*\*\*\*

|        |      |        |      |       |      |        |      |
|--------|------|--------|------|-------|------|--------|------|
| ACNT   | 025E | AD1A   | 0468 | ADD   | 043B | ADD0   | 0460 |
| ADD1   | 0468 | ADD10  | 0507 | ADD11 | 0511 | ADD2   | 046B |
| ADD3   | 048A | ADD3A  | 048D | ADD4  | 049C | ADD5   | 04B3 |
| ADD6   | 04CA | ADD7   | 04D4 | ADD9  | 04FD | ADRBIT | 0040 |
| ALIST  | 10A2 | AMEM   | 10FE | A0BJ  | 10A9 | ASM    | 105E |
| ASM2   | 107E | ASM2A  | 1098 | ASM3  | 109C | ASM3A  | 10B0 |
| ASM4   | 10DC | ASM5   | 1117 | ASM5A | 1124 | ASM6   | 1141 |
| ASM6AA | 1151 | ASTLIN | 1369 | BADR1 | 023D | BADR2  | 0241 |
| BFA    | 0245 | BFL    | 0529 | BFLA  | 0532 | BFR    | 0247 |



|        |      |        |      |        |      |        |      |
|--------|------|--------|------|--------|------|--------|------|
| BLKLIN | 1348 | CFLD   | 07B5 | CHI    | 0783 | CKZ2   | 18F0 |
| CKZ80  | 1809 | CKZRET | 18EE | CLASS  | 0784 | CLEAR  | 02EB |
| CLR2   | 02F3 | CNV2HX | 1385 | CNVDEC | 03DE | CNVLNO | 19A0 |
| CONVI  | 03D7 | CONVIA | 03E0 | CONV2  | 03ED | CPNT   | 0177 |
| CRLF   | 0189 | CZ6    | 18FA | CZ7    | 1914 | CZ8    | 1921 |
| DCNT   | 0260 | DDFDXY | 1FDA | DDFD22 | 1FE3 | DELETE | 03F2 |
| DL0    | 040F | DL1    | 0415 | DL2    | 0426 | DPOBJ2 | 236B |
| DPOBJ3 | 237C | DPOBJR | 233C | DSAVE  | 13A3 | DSPBIT | 0080 |
| DTCTCC | 0168 | EADR1  | 023F | EADR2  | 0243 | ECA    | 22E6 |
| ECH2   | 16C8 | ECHA   | 16C9 | ECHECK | 16AD | EJ2    | 14A4 |
| EJ3    | 14AF | EJ4    | 14C0 | EJECT  | 1497 | ENDFMT | 0752 |
| EOASM  | 2465 | EOB    | 0249 | EOLADR | 024D | ERCDZ  | 22D8 |
| ERLNE  | 22E2 | ERR    | 02E0 | ERRORS | 22AE | EUREKA | 07F8 |
| FAN1   | 1518 | FAN1A  | 151B | FAN2   | 1528 | FAN3   | 1530 |
| FAN8   | 1599 | FANE2  | 15A7 | FANEND | 1691 | FANEOL | 159C |
| FANE05 | 158A | FANLYZ | 150F | FANOP  | 16D4 | FASC   | 15B1 |
| FASC1  | 15B5 | FASC2  | 15C7 | FASC3  | 15D8 | FASCBT | 0020 |
| FBA    | 02AB | FBA1   | 02BB | FBA2   | 02C1 | FBA3   | 02CB |
| FBA4   | 02CE | FBA5   | 02D8 | FBUF   | 16D9 | FCFLAG | 22AA |
| FDC2   | 161E | FDCEND | 164E | FDECBT | 0004 | FDECM1 | 161D |
| FEND   | 16A5 | FHEXBT | 0002 | FHXDC  | 1657 | FHXDC0 | 1659 |
| FHXEND | 1688 | FHXOK  | 1677 | FIRST  | 024F | FLAG   | 0259 |
| FLCHAR | 22AB | FLGZ80 | 1F77 | FLNGTH | 16CB | FMRC0  | 1458 |
| FMRC02 | 1461 | FNRSLT | 16D7 | FOCT   | 15F0 | FOCTBT | 0008 |
| FOEND  | 1614 | FORMAT | 0621 | FPI    | 01EA | FPRM   | 01DD |
| FPRM1  | 01F3 | FPRM2  | 0200 | FRM1   | 062C | FRM10  | 070A |
| FRM11  | 0716 | FRM12  | 071A | FRM12A | 0729 | FRM13  | 074A |
| FRM2   | 0640 | FRM3   | 0680 | FRM4   | 0692 | FRM5   | 06A1 |
| FRM6   | 06C2 | FRM7   | 06D4 | FRM8   | 06EB | FRM9   | 06FE |
| FSRCD  | 1437 | FSRCD2 | 1449 | FSTC   | 16A9 | FSYM1  | 1574 |
| FSYMBL | 11F6 | FSYMBT | 0010 | FTYPE  | 16D3 | FUNDEF | 16CF |
| GDVCIN | 0186 | GTPCHR | 0171 | HANGUP | 1805 | HEX    | 0789 |
| HSAYE  | 0787 | IDC    | 01A7 | IDC2   | 01AA | IDCRTN | 01CF |
| IDIFF  | 0253 | IL0080 | 1F8C | ILC    | 1888 | INFLAG | 139F |
| INTRML | 015C | ITBL   | 0812 | KB     | 0262 | L2     | 0312 |
| L3     | 032D | L3A    | 0334 | L4     | 0337 | L5     | 037A |
| L5A    | 037D | L5A1   | 0384 | L5AB   | 0396 | L5B    | 03A8 |
| L5C    | 03AF | L6     | 03CA | L7     | 03D4 | LABEL  | 079D |
| LACHAR | 16D5 | LADR   | 0248 | LASTIN | 025A | LDRSIZ | 025B |
| LDVCIN | 0165 | LINZ   | 13BC | LIST   | 0311 | LNCNT  | 023B |
| LINTHT | 0002 | LP2    | 1488 | LP3    | 147B | LPNT   | 146C |
| LRAC0D | 1F17 | LSTFLG | 13A2 | LUOP   | 06A7 | MDFBIT | 0008 |
| MNEM   | 07A5 | MON    | 0100 | MON1   | 0116 | MON2   | 0119 |
| NCMPS  | 1387 | NE2    | 121E | NEGHL  | 0433 | NEMSG  | 1224 |
| NLACHR | 16D6 | NOEND  | 121B | NOFILL | 1B7C | NOPROG | 14F1 |
| NPMSG  | 14FA | NSWAPS | 13B5 | NSYMS  | 13AD | NULLS  | 025F |
| NXT    | 0446 | NXTCHR | 0255 | OBFLAG | 13A1 | OCODE  | 22B0 |
| OP     | 080E | OPBIT  | 0020 | OPCMP1 | 0802 | OPCMR  | 07FE |
| OPNDF  | 07AB | OPSRC  | 07D2 | OPSRC1 | 07DF | ORGVAL | 18BA |
| OUT    | 0180 | PIASC  | 1851 | PIASC2 | 185E | P1B    | 1721 |
| P1B1   | 1757 | P1B2   | 1768 | P1C    | 1775 | P1C0   | 17C0 |
| P1C2   | 1CC1 | P1D    | 1799 | P1DTA  | 186D | P1DTA1 | 1873 |
| P1DTA2 | 1888 | P1DTA3 | 189A | P1E    | 179F | P1EMSG | 1950 |
| P1EQUR | 1956 | P1ERFL | 192E | P1INCI | 18A0 | P1INC2 | 18A6 |
| P1INC3 | 18AC | P1INC4 | 18B2 | P1LERR | 197E | P1LTBL | 18C0 |
| P1ORG  | 180E | P1ORG2 | 181B | P1QMSG | 1978 | P1RES  | 183F |
| P1RES2 | 184D | P1SUM  | 17B6 | P1TITL | 1827 | P1TMP  | 18BC |
| P1TTL2 | 182F | P2A    | 19D7 | P2A1   | 1A14 | P2A1A  | 1A1F |

|        |      |        |      |        |      |        |      |
|--------|------|--------|------|--------|------|--------|------|
| P2A2   | 1C31 | P2ARS  | 1ADA | P2ASC  | 1BFA | P2ASC2 | 1C1E |
| P2ASC3 | 1C26 | P2ASC4 | 1C2C | P2ASC5 | 1C37 | P2B    | 1A35 |
| P2BDOP | 21E5 | P2C0   | 1A57 | P2C1   | 1CA7 | P2C10  | 20AB |
| P2C10B | 20C0 | P2C11  | 20D6 | P2C11A | 20DF | P2C12  | 20EB |
| P2C13  | 20F5 | P2C14  | 20FF | P2C15  | 2129 | P2C16  | 2153 |
| P2C17  | 2181 | P2C18  | 21C8 | P2C18A | 21CF | P2C19  | 21DB |
| P2C1A  | 1CCE | P2C1AA | 1CD2 | P2C1B  | 1CD5 | P2C1C  | 1CDB |
| P2C2   | 1CE6 | P2C3   | 1D10 | P2C3A  | 1D47 | P2C3B  | 1D51 |
| P2C3C  | 1D56 | P2C3D  | 1D5E | P2C4   | 1D66 | P2C4A  | 1D7D |
| P2C4B  | 1D83 | P2C5   | 1DC6 | P2C5AA | 1E02 | P2C5B  | 1E05 |
| P2C5C  | 1E08 | P2C5D  | 1E10 | P2C5E  | 1E16 | P2C5F  | 1E23 |
| P2C6   | 1E2B | P2C6A  | 1E47 | P2C6A1 | 1E4A | P2C6B  | 1E4B |
| P2C6C  | 1E79 | P2C6D  | 1E8B | P2C7   | 1E97 | P2C8   | 1EA2 |
| P2C8A  | 1EC2 | P2C8B  | 1ECB | P2C8D  | 1ED7 | P2C8E  | 1EDC |
| P2C8F  | 1EE3 | P2C9   | 1EEB | P2C9A  | 1F00 | P2C9B  | 1F03 |
| P2CA   | 1F1B | P2CAB  | 1F38 | P2CAE  | 1F3E | P2CB   | 1F49 |
| P2CB1  | 1F59 | P2CC   | 1F8D | P2CC2  | 1FD4 | P2CD   | 2031 |
| P2CE   | 203F | P2CE2  | 2055 | P2CF   | 206E | P2CF2  | 2098 |
| P2CF3  | 20A5 | P2DATA | 1BA9 | P2DBL  | 1C4B | P2DBL2 | 1C6C |
| P2DBL3 | 1C75 | P2DBL4 | 1C7A | P2DBL5 | 1C83 | P2DC   | 1BEC |
| P2DTA2 | 1BCB | P2DTA3 | 1BE7 | P2DTA4 | 1BF2 | P2DTAB | 1BAF |
| P2DTND | 1BE1 | P2END  | 1AD7 | P2EQU  | 1A9C | P2JTBL | 1B05 |
| P2LNO  | 22AC | P2LNTH | 22AF | P2LRA  | 1F09 | P2LST  | 1AF7 |
| P2PTR  | 22E0 | P2ORG  | 1AB0 | P2ORG2 | 1ABD | P2ORG3 | 1ACC |
| P2PP   | 1E82 | P2PP1  | 1E67 | P2PRA  | 221D | P2PRB  | 2224 |
| P2PRC  | 2232 | P2PRCA | 225C | P2PRD  | 2263 | P2PRF  | 2276 |
| P2PRG  | 2241 | P2PRH  | 224B | P2PRI  | 2286 | P2PRNT | 21F5 |
| P2REM  | 1C3F | P2RES  | 1B37 | P2RESL | 1B9E | P2RESU | 1B82 |
| P2RSUF | 1B93 | P2TCA  | 1F0F | P2TPP  | 1E54 | P2UNL  | 1AF0 |
| P2UNLA | 1AF1 | P2ZAER | 2021 | P2ZDER | 2029 | P2ZL2  | 200B |
| P2ZLER | 2006 | P2ZRER | 2019 | P2ZUER | 2011 | PAGE   | 13BB |
| PASS1  | 170F | PASS2  | 19C6 | PCHBL2 | 22FA | PCHBL3 | 2301 |
| PCHBLT | 22F1 | PCHCHR | 017A | PCHFRM | 23D8 | PCHOB2 | 2327 |
| PCHOB3 | 2330 | PCHOBJ | 2312 | PCHRES | 238D | PCHRS2 | 23BB |
| PCHRS3 | 23CC | PD1    | 13C2 | PD2    | 13C3 | PDVC   | 0000 |
| PDVINT | 0162 | PFCH   | 01D0 | PFLAG  | 025D | PGN8   | 14C9 |
| PGN81  | 14D2 | PGN82  | 14DB | PGNO   | 13BD | PRFLAG | 13A0 |
| PRT    | 0199 | PRT2   | 019C | PSY1   | 12E3 | PSY2   | 12F0 |
| PSY2A  | 12FB | PSY3   | 12FD | PSYEND | 133D | PSYLN  | 1341 |
| PSYMTB | 12B8 | PTRLR  | 229B | PUNCH  | 0309 | R2     | 0550 |
| R3     | 05F6 | R3A    | 05FA | R4     | 060D | R5     | 0616 |
| R6     | 055D | RCDBLK | 23E5 | RCDORG | 23E0 | RCDPTR | 23E3 |
| RCD5I2 | 23E2 | RCH    | 0599 | RCH2   | 05B6 | RDPMSG | 115E |
| RDRINT | 015F | RDRSTP | 016E | RDRSTT | 016B | READ   | 053E |
| REBIT  | 0010 | REWIND | 0183 | RKB    | 017D | RSLNTH | 22EF |
| RSTRD  | 11F0 | RTMON  | 22A4 | RTNFLG | 0786 | RTR    | 05C5 |
| RTRB   | 05D6 | RTRCZ  | 05E2 | SB     | 0174 | SDRF   | 1D8E |
| SDRFB  | 1D81 | SDRFC  | 1DC4 | SDRFD  | 1DAE | SE2    | 1192 |
| SECOND | 0251 | SKEL   | 0785 | SLL2   | 16F2 | SLL3   | 170B |
| SLLBL  | 16DF | SPTR   | 13A7 | SPTR2  | 13A9 | SRCPTR | 13B9 |
| SRCSTB | 11B0 | SFR    | 1C8B | SFR1   | 1C90 | SFR2   | 1C9B |
| SFILE  | 1C9F | SRST1  | 11C6 | SRST2  | 11D1 | SRT2A  | 1278 |
| SRTST  | 1239 | SRTST0 | 1242 | SRTST1 | 1254 | SRTST2 | 1265 |
| SRTST3 | 129C | SRTSWP | 1281 | STBMSG | 134D | STFILL | 1B73 |
| STLOC  | 13A5 | STMT   | 11EF | STO    | 11FF | STOM   | 1205 |
| STPCOD | 1F15 | SVAL   | 13AB | SWAP2  | 128A | SYMBIT | 0004 |
| SYMCNT | 13B3 | SYMENT | 1171 | SYMLOC | 13AF | SYMPTR | 13B1 |
| TBUF   | 13EF | TCACOD | 1F19 | TCHK1  | 1FE5 | TCHK2  | 1FF2 |

PAGE 77      Z80 RESIDENT EDITOR/ASSEMBLER

|        |      |        |      |        |      |        |      |
|--------|------|--------|------|--------|------|--------|------|
| TEMP   | 0257 | TITLE  | 1308 | TLFLAG | 188E | TR     | 065F |
| TRLSIZ | 025C | TSIZE  | 0162 | UNDFBT | 0001 | XCF    | 0735 |
| XCLI   | 0773 | XCLINE | 076E | XMTTL  | 0759 | XMTTL2 | 075E |
| XMTTLE | 076B | XOP    | 0671 | XOPI   | 067B | ZH2    | 139C |
| ZHEX   | 1393 | ZL7C1  | 1FFE | ZL7C2  | 2000 | ZL7CHK | 1FF4 |

## APPENDIX C / THE ZEDBUG DEBUGGING MONITOR

When the ZEDBUG object tape has been loaded successfully as per the instructions in appendix A it must be initialized to operate with your peripheral configuration. This appendix describes the necessary steps in this initialization.

The ZEDBUG monitor will perform all of its functions with as little peripheral equipment as a Teletype, though it is capable of taking advantage of other peripherals if they are available. The initialization consists of supplying a series of up to eleven input/output routines, beginning at memory location :3E70 after the end of ZEDBUG proper, and filling in a jump table which begins at location :3660 within ZEDBUG. The program itself begins at location :3600, allowing operation within a 16K machine. If more memory is available the program can be easily reassembled on any required origin.

Each of the eleven possible I/O routines must be in subroutine form, i.e., it must end with a **RET** instruction (:C9). If a routine is not required either because the peripheral it services does not exist in your system or because its function is irrelevant it may simply be ignored, though the corresponding position in the I/O jump table must be filled in with a **RET** instruction. In a system equipped only with an ALTAIR or IMSAI style Teletype without reader control five of the eleven routines become meaningless. The routines and their functions are:

- |        |                                                                                                                                   |
|--------|-----------------------------------------------------------------------------------------------------------------------------------|
| INTMNL | Initialize the terminal interface.                                                                                                |
| INRDR  | Initialize the paper tape reader interface.                                                                                       |
| GTFRM  | Read one tape frame from the paper tape read device and return with the frame read in the A register. Disturb no other registers. |

|        |                                                                                                                                                                                                                                                                                                                                                                              |
|--------|------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------|
| RDRSTT | Start the paper tape reader.                                                                                                                                                                                                                                                                                                                                                 |
| RDRSTP | Stop the paper tape reader.                                                                                                                                                                                                                                                                                                                                                  |
| RKB    | Read one character from the terminal keyboard and return with it in the A register. The character must be in upper case ASCII with the high bit forced on. Upper case ASCII can be forced by testing the character for the :E1 - :FA range after the high bit is forced to one. If the character is in this range execute <b>ANI :DF</b> . Disturb no register other than A. |
| OUT    | Print the character in the A register on the system terminal device. Wait for the character to clear and then return with no register disturbed.                                                                                                                                                                                                                             |
| PDVCIN | Initialize paper tape punch device interface.                                                                                                                                                                                                                                                                                                                                |
| PCHCHR | Send the character in A to the paper tape punch. Return with no register disturbed.                                                                                                                                                                                                                                                                                          |
| DTCTCC | Detect control-C typed. If control-C was typed return with the carry bit on. If no character was typed or if something other than control-C was typed return with carry off. Do not wait for something to be typed, simply test to see whether something has already been typed. Only A may be changed. The code for control-C is 83 <sub>16</sub> .                         |
| SB     | Wait for space bar. Wait until a character is typed. Return when the space bar code (A0 <sub>16</sub> ) is typed. If something other than space is typed ignore it and continue to wait for the space. Do <i>not</i> echo the character typed, space or otherwise, to the terminal.                                                                                          |

When the above routines have been entered into memory beginning at :3E70 the jump table at :3660 must be filled in, in the order shown below. If a routine does not exist, fill in the three word jump table entry with C9 00 and 00. A jump instruction is C3. When fill-



ing in the jump address, remember that the low order half of the address occupies the low order memory word. If the INTMNL routine began at :3E70 the 3 word jump table entry at :3660 would be:

```
3660    C3
3661    70
3662    3E
```

The order of the jump table entries, beginning at :3660, is:

| <i>address</i> | <i>instruction</i> |
|----------------|--------------------|
| 3660           | JMP INTMNL         |
| 3663           | JMP INRDR          |
| 3666           | JMP GTFRM          |
| 3669           | JMP RDRSTT         |
| 366C           | JMP RDRSTP         |
| 366F           | JMP RKB            |
| 3672           | JMP OUT            |
| 3675           | JMP PDVCIN         |
| 3678           | JMP PCHCHR         |
| 367B           | JMP DTCTCC         |
| 367E           | JMP SB             |

This process is really a good deal simpler than it looks. For a system with only a Teletype without reader control the table looks like this:

|      |     |       |                                         |
|------|-----|-------|-----------------------------------------|
| 3660 | RET |       | irrelevant                              |
|      | DBL | 0     | fill with zeros                         |
| 3663 | RET |       | irrelevant                              |
|      | DBL | 0     | fill with zeros                         |
| 3666 | JMP | GTFRM | get paper tape frame<br>from TTY reader |
| 3669 | RET |       | irrelevant                              |
|      | DBL | 0     | fill with zeros                         |
| 366C | RET |       | irrelevant                              |
|      | DBL | 0     | fill with zeros                         |
| 366F | JMP | RKB   | get character from<br>keyboard          |

|      |     |        |                                                                                         |
|------|-----|--------|-----------------------------------------------------------------------------------------|
| 3672 | JMP | OUT    | print character in A                                                                    |
| 3675 | RET |        | irrelevant                                                                              |
|      | DBL | 0      | fill with zeros                                                                         |
| 3678 | JMP | OUT    | punch character. For<br>this simple configura-<br>tion print and punch are<br>the same. |
| 367B | JMP | DTCTCC | detect control-C                                                                        |
| 367E | JMP | SB     | wait for space bar                                                                      |

As a guide to initializing for more elaborate sets of peripherals, the ZEDBUG initialization for the writer's system is shown in assembled form immediately before the ZEDBUG assembly listing in this appendix.

When all of the above adjustments are made, transfer control to the first location of ZEDBUG, :3600. It should respond with a prompt character (#). If it does not there is some error in your I/O routines or in the jump table. Go back and correct it. When ZEDBUG responds with the prompt, type

D3600 nnnn.

Where nnnn is the last location used by your I/O routines. The computer will pause while you turn on the punch device. Strike the space bar and initialized copy of ZEDBUG, complete with your I/O routines will be punched on the system punch device. If this is not successful there is some error in your punch service routines.

You may now use the ZEDBUG load command (L.) to load the object copy of the assembler. Put the tape in the reader and type L. The period after L is part of the command. If the tape reader must be manually started, as with a Teletype without reader control, set the switch to the start position. The tape should begin running through the reader. If it does not, there is something wrong with your INRDR or RDRSTT routines.

When the tape has finished passing through the reader it should stop if you have reader control, and one of two things will happen. Either a prompt character (#) will appear, meaning that the load is successful or the word:

CKSM

will appear, meaning that an error has occurred. If you get the CKSM message either there is an error in your GTFRM routine or a hardware malfunction exists in the paper tape reader. It is also possible that the tape has become dirty through dragging on the floor during the load process. If your system has no reader control the tape will continue to slew through the reader. On a Teletype this will cause a series of meaningless “?” characters to be printed. This does no harm and is due to the fact that the computer cannot tell whether the character it is seeing is from the reader or the keyboard. It is expecting input from the keyboard and the null frames still being read from the tape constitute illegal commands. The “?” is ZEDBUG’s way of protesting an illegal command.

If your system is equipped with a disk, now is the time to copy ZEDBUG to the disk. This will save the tedious job of reinitializing if tapes are lost or destroyed. As with any important piece of software, make a backup copy.

You are now ready to initialize the assembler in appendix B.

PAGE 01 I/O ASSIGNMENTS FOR ZEDBUG

```

002      *
003      * I/O ROUTINES FOR THE WRITER'S SYSTEM
004      * SUPPORTING THE FOLLOWING PERIPHERALS:
005      *
006      * 1 - CRT TERMINAL
007      * 2 - REMEX HIGH SPEED TAPE READER
008      * 3 - REMEX HIGH SPEED TAPE PUNCH
009      * 4 - OKIDATA LINE PRINTER
010      *

```

I/O JUMP TABLE

```

011
012
013
014      ORG      :3660
015 3660 C3703E      JMP      INTMNL
016 3663 C3793E      JMP      INRDR
017 3666 C38F3E      JMP      GTFRM
018 3669 C3973E      JMP      RDRSTT
019 366C C39C3E      JMP      RDRSTP
020 366F C3A13E      JMP      RKB
021 3672 C3B43E      JMP      OUT
022 3675 C3BF3E      JMP      PDVCIN
023 3678 C3D03E      JMP      PCHCHR
024 367B C3EA3E      JMP      DTCTCC
025 367E C3FA3E      JMP      SB

```

I/O SERVICE ROUTINES

```

026
027
028
029      ORG      :3E70      THESE ROUTINES MAY BE IN ANY
030                                ORDER BUT THEY MUST BEGIN
031                                BEYOND THE END OF ZEDBUG
032                                ITSELF (:3E6C)
033

```

```

034
035 3E70 3E03      INTMNL  MVI      A,3      INITIALIZE TERMINAL
036 3E72 D320      OUT      STAT      DEVICE INTERFACE
037 3E74 3E11      MVI      A,:11
038 3E76 D320      OUT      STAT
039 3E78 C9        RET
040      STAT      EQU      :20      DEVICE ADDRESS
041      DATA      EQU      :21      ASSIGNMENT
042

```

```

043
044 3E79 AF        INRDR   ZAR                INITIALIZE PAPER
045 3E7A D310      OUT      CTLA      TAPE READER
046 3E7C D312      OUT      CTLB      INTERFACE
047 3E7E D313      OUT      DTAB
048 3E80 3D        DCR      A
049 3E81 D311      OUT      DTAA
050 3E83 3E04      MVI      A,4
051 3E85 D310      OUT      CTLA
052 3E87 3E06      MVI      A,6
053 3E89 D312      OUT      CTLB
054 3E8B CD9C3E    CALL     RDRSTP
055 3E8E C9        RET
056

```

```

057
058 3E8F DB12      GTFRM   IN      CTLB      READ ONE TAPE
059 3E91 17        RAL                FRAME AND RETURN

```

PAGE 02 I/O ASSIGNMENTS FOR ZEDBUG

|     |           |    |        |       |                            |
|-----|-----------|----|--------|-------|----------------------------|
| 060 | 3E92 30FB | \$ | BNC    | GTFRM |                            |
| 061 | 3E94 DB13 |    | IN     | DTAB  |                            |
| 062 | 3E96 C9   |    | RET    |       |                            |
| 063 |           |    |        |       |                            |
| 064 |           |    |        |       |                            |
| 065 | 3E97 3E0F |    | RDRSTT | MVI   | A, GOBITS START PAPER TAPE |
| 066 | 3E99 D311 |    |        | OUT   | DTAB READER                |
| 067 | 3E9B C9   |    |        | RET   |                            |
| 068 |           |    |        |       |                            |
| 069 |           |    |        |       |                            |
| 070 | 3E9C 3E00 |    | RDRSTP | MVI   | A, STPBTS STOP PAPER TAPE  |
| 071 | 3E9E D311 |    |        | OUT   | DTAB READER                |
| 072 | 3EA0 C9   |    |        | RET   |                            |
| 073 |           |    |        |       |                            |
| 074 |           |    | CTLA   | EQU   | :10 PAPER TAPE READER      |
| 075 |           |    | DTAB   | EQU   | CTLA+1 DEVICE ADDRESS      |
| 076 |           |    | CTLB   | EQU   | DTAB+1 DEFINITIONS         |
| 077 |           |    | DTAB   | EQU   | CTLB+1                     |
| 078 |           |    | GOBITS | EQU   | :F                         |
| 079 |           |    | STPBTS | EQU   | :F0                        |
| 080 |           |    |        |       |                            |
| 081 |           |    |        |       |                            |
| 082 | 3EA1 DB20 |    | RKB    | IN    | STAT READ ONE CHARACTER    |
| 083 | 3EA3 1F   |    | RAR    |       | FROM TERMINAL              |
| 084 | 3EA4 30FB | \$ | BNC    | RKB   | KEYBOARD AND               |
| 085 | 3EA6 DB21 |    | IN     | DATA  | ECHO BACK TO               |
| 086 | 3EA8 F680 |    | ORI    | :80   | TERMINAL. FORCE            |
| 087 |           |    |        |       | THE HIGH BIT TO ONE.       |
| 088 | 3EAA FEE1 |    | CPI    | :E1   | CHECK FOR LOWER            |
| 089 | 3EAC 3806 | \$ | BLS    | OUT   | CASE ASCII                 |
| 090 | 3EAE FE7B |    | CPI    | :7B   | ...                        |
| 091 | 3EB0 3002 | \$ | BGE    | OUT   |                            |
| 092 | 3EB2 E6DF |    | ANI    | :DF   | ZAP LOWER CASE BIT         |
| 093 |           |    |        |       |                            |
| 094 |           |    |        |       |                            |
| 095 | 3EB4 D321 |    | OUT    | DATA  | SEND CHARACTER             |
| 096 | 3EB6 F5   |    |        | PUSH  | PSW IN A TO TERMINAL       |
| 097 | 3EB7 DB20 |    | OUT2   | IN    | STAT                       |
| 098 | 3EB9 1F   |    | RAR    |       |                            |
| 099 | 3EBA 1F   |    | RAR    |       |                            |
| 100 | 3EBB 30FA | \$ | BNC    | OUT2  |                            |
| 101 | 3EBD F1   |    | POP    | PSW   | RETURN WITH A              |
| 102 | 3EBE C9   |    | RET    |       | UNCHANGED                  |
| 103 |           |    |        |       |                            |
| 104 |           |    |        |       |                            |
| 105 | 3EBF AF   |    | PDVCIN | ZAR   | INITIALIZE PAPER           |
| 106 | 3EC0 D314 |    |        | OUT   | CTA TAPE PUNCH             |
| 107 | 3EC2 D316 |    |        | OUT   | CTB INTERFACE              |
| 108 | 3EC4 3D   |    |        | DCR   | A                          |
| 109 | 3EC5 D317 |    |        | OUT   | DTB                        |
| 110 | 3EC7 3E04 |    |        | MVI   | A, 4                       |
| 111 | 3EC9 D314 |    |        | OUT   | CTA                        |
| 112 | 3ECB 3E36 |    |        | MVI   | A, :36                     |
| 113 | 3ECD D316 |    |        | OUT   | CTB                        |
| 114 | 3ECF C9   |    |        | RET   |                            |
| 115 |           |    |        |       |                            |
| 116 |           |    |        |       |                            |
| 117 | 3ED0 D317 |    | PCHCHR | OUT   | DTB PUNCH FRAME IN A       |



PAGE 03 I/O ASSIGNMENTS FOR ZEDBUG

|               |        |     |        |                       |
|---------------|--------|-----|--------|-----------------------|
| 118 3ED2 DB15 | PCH2   | IN  | DTA    | ON PAPER TAPE         |
| 119 3ED4 E681 |        | ANI | :81    |                       |
| 120 3ED6 20FA | \$     | BNZ | PCH2   |                       |
| 121 3ED8 3E3E |        | MVI | A, :3E |                       |
| 122 3EDA D316 |        | OUT | CTB    |                       |
| 123 3EDC 3E36 |        | MVI | A, :36 |                       |
| 124 3EDE D316 |        | OUT | CTB    |                       |
| 125 3EE0 DB17 |        | IN  | DTB    |                       |
| 126 3EE2 DB16 | PCH3   | IN  | CTB    |                       |
| 127 3EE4 17   |        | RAL |        |                       |
| 128 3EE5 30FB | \$     | BNC | PCH3   |                       |
| 129 3EE7 DB17 |        | IN  | DTB    |                       |
| 130 3EE9 C9   |        | RET |        |                       |
| 131           | CTA    | EQU | :14    | PUNCH DEVICE          |
| 132           | DTA    | EQU | CTA+1  | ADDRESS               |
| 133           | CTB    | EQU | DTA+1  | ASSIGNMENTS           |
| 134           | DTB    | EQU | CTB+1  |                       |
| 135           |        |     |        |                       |
| 136           |        |     |        |                       |
| 137 3EEA DB20 | DTCTCC | IN  | STAT   | DETECT CONTROL-C      |
| 138 3EEC 1F   |        | RAR |        |                       |
| 139 3EED D0   |        | RNC |        | RETURN IMMEDIATELY    |
| 140           |        |     |        | IF NOTHING TYPED!     |
| 141 3EEE DB21 |        | IN  | DATA   |                       |
| 142 3EF0 F680 |        | ORI | :80    | FORCE HIGH BIT ON     |
| 143 3EF2 FE83 |        | CPI | :83    |                       |
| 144 3EF4 2002 | \$     | BNE | DTCT2  |                       |
| 145 3EF6 37   |        | STC |        | CARRY ON IF CONTROL-C |
| 146 3EF7 C9   |        | RET |        | TYPED, RETURN         |
| 147 3EF8 B7   | DTCT2  | RSC |        | CARRY OFF OTHERWISE   |
| 148 3EF9 C9   |        | RET |        |                       |
| 149           |        |     |        |                       |
| 150           |        |     |        |                       |
| 151 3EFA DB20 | SB     | IN  | STAT   | WAIT FOR SPACE BAR    |
| 152 3EFC 1F   |        | RAR |        |                       |
| 153 3EFD 30FB | \$     | BNC | SB     | DO NOT RETURN         |
| 154 3EFF DB21 |        | IN  | DATA   | UNTIL SPACE IS TYPED  |
| 155 3F01 F680 |        | ORI | :80    |                       |
| 156 3F03 FEA0 |        | CPI | /'     |                       |
| 157 3F05 20F3 | \$     | BNE | SB     |                       |
| 158 3F07 C9   |        | RET |        |                       |
| 159 3F08      |        | END |        |                       |

\*\*\*\*\*  
 \* S Y M B O L   T A B L E \*  
 \*\*\*\*\*

|        |      |        |      |        |      |        |      |
|--------|------|--------|------|--------|------|--------|------|
| CTA    | 0014 | CTB    | 0016 | CTLA   | 0010 | CTLB   | 0012 |
| DATA   | 0021 | DTA    | 0015 | DTA    | 0011 | DTAB   | 0013 |
| DTB    | 0017 | DTCT2  | 3EF8 | DTCTCC | 3EEA | GOBITS | 000F |
| GTFRM  | 3EBF | INRDR  | 3E79 | INTMNL | 3E70 | OUT    | 3EB4 |
| OUT2   | 3EB7 | PCH2   | 3ED2 | PCH3   | 3EE2 | PCHCHR | 3ED0 |
| POVCIN | 3EBF | RDRSTP | 3E9C | RDRSTT | 3E97 | RKB    | 3EAI |
| SB     | 3EFA | STAT   | 0020 | STPBT5 | 00F0 |        |      |

## SUMMARY OF ZEDBUG COMMANDS

Except as noted below, all numbers typed in by the user are hexadecimal. The symbol @ is used below to indicate a space typed by the user. The symbols aaaa and bbbb indicate 4 digit hexadecimal numbers, nn and dd are two digit hexadecimal numbers. The symbol rp means the higher register of a register pair.

|             |                                                                                                                                                                                                                                                                                                                                                    |
|-------------|----------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------|
| Maaaa.      | Examine location aaaa. If nn is typed in response the contents of aaaa will be changed to nn. Typing space displays the next higher memory address, typing comma the next lower memory address. A period terminates the command. A hexadecimal number typed before the period, blank or comma replaces the contents of aaaa with the typed number. |
| Maaaa@bbbb. | Displays the entire block aaaa through bbbb on the system terminal device.                                                                                                                                                                                                                                                                         |
| MS.         | Displays contents of the last item pushed onto the user stack. Space bar and comma works as for Maaaa above.                                                                                                                                                                                                                                       |
| /D. or /E.  | Causes interrupts to be disabled or enabled during execution of jump or breakpoint commands.                                                                                                                                                                                                                                                       |
| R.          | Display all pseudo registers.                                                                                                                                                                                                                                                                                                                      |
| Rrp@        | Display pseudo register pair rp. Contents of rp are changed by typing a hexadecimal value. Command terminates with a period.                                                                                                                                                                                                                       |
| RA@         | Display pseudo A register. Modification as for Rrp above.                                                                                                                                                                                                                                                                                          |
| Rrp'        | Display/modify prime pseudo register pair.                                                                                                                                                                                                                                                                                                         |

- RA'                      Display/modify pseudo A register.
- RF@                     Displays individual bits of the pseudo F register. Bits are changed by typing name of bit, S, Z, A, V, N or C followed by one or zero. Period terminates the command.
- RF'                     Display/modify prime pseudo F register, as for RF@ above.
- L.                       Load assembler-generated object tape or output of D command below from the system tape reader. Error in load generates the message CKSM.
- Daaaa@bbbb.           Punch memory addresses aaaa through bbbb onto paper tape. The punching is preceded and followed by a pause to turn punch device on and off. Space bar resumes activity.
- Jaaaa.                   Load machine registers from pseudo registers and transfer control to location aaaa.
- Baaaa@bbbb.           Sets a breakpoint at bbbb, loads machine registers from pseudo registers and transfers control to aaaa. Upon return from breakpoint all pseudo registers are reset to values of machine registers, then typed out as for R. command above. If control does not return stop the computer, set PC to beginning of ZEDBUG and start. This will clear the breakpoint in the program being tested and product the BKPT FAILED message.
- Faaaa@bbbb@nn.       Fills locations aaaa through bbbb with the value nn. Value of aaaa must be less than that of bbbb.
- Saaaa@bbbb@nn.       Searches locations aaaa through bbbb for the value nn. Addresses of locations containing nn are displayed on terminal device.

- Inn.** Reads device address nn and displays result. Space bar repeats the read and display. Period terminates the command.
- Onn@dd.** Output the data dd to device address nn. Space repeats the output. Period terminates the command.

M or S commands may be terminated at any time by control-C.

PAGE 01 <<<< Z E D B U G M O N I T O R >>>>

```

002 *****
003 * HEXADECIMAL DEBUG FOR Z80 BASED SYSTEMS *
004 *****
005 *
006 * W.W. 6/23/78
007 *
008
009 3600 F3 ZEDBUG ORG :3600 KILL INTERRUPTS
010 3601 310036 LXI SP,ZEDBUG
011 3604 CD6036 CALL INTMNL
012 3607 CD6336 CALL INRDR
013 360A CD7536 CALL PDVCIN
014 360D 3A4939 LDA ILOC
015 3610 B7 ORA A CHECK FOR BREAKPOINT ON
016 3611 C2CF39 JNZ BPFail CLEAR THE BREAKPOINT
017 3614 CD7637 RESUME CALL CRLF** RETURN CARRIAGE
018 3617 3EA3 MVI A,'#' SEND PROMPT CHARACTER
019 3619 CD7236 CALL OUT***
020 361C CD6F36 CALL RKB***
021 361F FECD CPI 'M' MEMORY INSPECT/MODIFY
022 3621 CAB236 JEQ INS INTERRUPT ENABLE FLAG SET
023 3624 FEAR CPI '/'
024 3626 CA393A JEQ IE REGISTER INSPECT/MODIFY
025 3629 FED2 CPI 'R'
026 362B CA573A JEQ R LOAD ASSEMBLER OBJECT OUTPUT
027 362E FECC CPI 'L'
028 3630 CA233C JEQ LOADER DUMP MEMORY TO TAPE
029 3633 FEC4 CPI 'D'
030 3635 CABA3C JEQ DUMP JUMP
031 3638 FECA CPI 'J'
032 363A CA0439 JEQ JUMP BREAKPOINT
033 363D FEC2 CPI 'B'
034 363F CA7339 JEQ XEQ FILL MEMORY BLOCK
035 3642 FEC6 CPI 'F'
036 3644 CAE93D JEQ FILL SEARCH MEMORY BLOCK
037 3647 FED3 CPI 'S'
038 3649 CAF73D JEQ SEARCH
039 364C FEC9 CPI 'I' INPUT FROM DEVICE
040 364E CADA3B JEQ INPUT OUTPUT TO DEVICE
041 3651 FECF CPI 'O'
042 3653 CAFE3B JEQ OUTPUT
043 3656 3EBF TILT MVI A,'?'
044 3658 310036 LXI SP,ZEDBUG
045 365B CD7236 CALL OUT***
046 365E 18B4 $ BRA RESUME
047 *
048 * I/O VECTOR
049 *
050 3660 C36036 INTMNL JMP INTMNL INITIALIZE TERMINAL
051 3663 C36336 INRDR JMP INRDR INITIALIZE TAPE READER
052 3666 C36636 GTFRM JMP GTFRM GET FRAME FROM READER
053 3669 C36936 RDRSTT JMP RDRSTT START READER
054 366C C36C36 RDRSTP JMP RDRSTP STOP READER
055 366F C36F36 RKB*** JMP RKB*** READ TERMINAL KEYBOARD
056 3672 C37236 OUT*** JMP OUT*** SEND CHAR. TO TERMINAL
057 3675 C37536 PDVCIN JMP PDVCIN INITIALIZE PUNCH
058 3678 C37836 PCHCHR JMP PCHCHR PUNCH FRAME IN A
059 367B C37B36 DTCTCC JMP DTCTCC DETECT CONTROL-C

```



```

060 367E C37E36      SB      JMP      SB      WAIT FOR SPACE BAR
061                  *
062                  * EXAMINE STACK CONTENTS
063                  *
064 3681 FED3          CSTACK CPI      'S'
065 3683 20D1          $      BNE      TILT
066 3685 CD6F36        CALL      RKB***
067 3688 FEAE          CPI      ' '
068 368A 20CA          $      BNE      TILT
069 368C 2A3E39        LHL      STKP
070 368F CD7637        CSTACK CALL      CRLF**
071 3692 7E            MOV      A,M
072 3693 47            MOV      B,A
073 3694 23            INX      H
074 3695 7E            MOV      A,M
075 3696 CDCD3B        CALL      PW
076 3699 78            MOV      A,B
077 369A CDCD3B        CALL      PW
078 369D 2B            DCX      H
079 369E CD6F36        CALL      RKB***
080 36A1 FEAE          CPI      ' '
081 36A3 CA1436        JEQ      RESUME
082 36A6 FEAC          CPI      ' '
083 36A8 2004          $      BNE      CSTACK3
084 36AA 23            INX      H
085 36AB 23            INX      H
086 36AC 18E1          $      BRA      CSTACK2
087 36AE 2B            CSTACK3 DCX      H
088 36AF 2B            DCX      H
089 36B0 18DD          $      BRA      CSTACK2
090                  *
091                  * MEMORY INSPECTION/MODIFICATION MACHINERY
092                  *
093 36B2 CDD83D        INS      CALL      FPRM
094 36B5 28CA          $      BZ      CSTACK
095 36B7 FEAE          CPI      ' '
096 36B9 CA663D        JEQ      PBLK
097 36BC FEAE          CPI      ' '
098 36BE C25636        JNE      TILT
099 36C1 CD7637        INS1     CALL      CRLF**
100 36C4 7C            MOV      A,H
101 36C5 CDCD3B        CALL      PW
102 36C8 7D            MOV      A,L
103 36C9 CDCD3B        CALL      PW
104 36CC 3EA0          MVI      A,' '
105 36CE CD7236        CALL      OUT***
106 36D1 7E            MOV      A,M
107 36D2 CDCD3B        CALL      PW
108 36D5 3EA0          MVI      A,' '
109 36D7 CD7236        CALL      OUT***
110 36DA 226737        SHLD     SAVE
111 36DD CDD83D        CALL      FPRM
112 36E0 2006          $      BNZ      INS3
113 36E2 2A6737        LHL      SAVE
114 36E5 C3FD36        JMP      MAYBE
115 36E8 3A6637        INS3     LDA      LASTIN
116 36EB FEAE          CPI      ' '
117 36ED 2809          $      BEQ      INS3A

```

RETURN CARRIAGE  
 MOVE HIGH ADDRESS  
 CONVERT AND PRINT  
 CONVERT LOW ADDRESS  
 CONVERT AND PRINT  
 PRINT BLANK  
 GET MEMORY CONTENTS  
 CONVERT AND PRINT  
 SAVE POINTER  
 NO. RELOAD POINTER  
 LEGITIMATE TERMINATOR?

|     |      |        |       |      |        |                       |
|-----|------|--------|-------|------|--------|-----------------------|
| 118 | 36F  | FEAC   |       | CPI  | ','    |                       |
| 119 | 36F1 | 2805   | \$    | BEQ  | INS3A  |                       |
| 120 | 36F3 | FEAE   |       | CPI  | ','    |                       |
| 121 | 36F5 | C25636 |       | JNE  | TILT   |                       |
| 122 | 36F8 | 7D     | INS3A | MOV  | A,L    |                       |
| 123 | 36F9 | 2A6737 |       | LHLD | SAVE   | RELOAD MEMORY POINTER |
| 124 | 36FC | 77     |       | MOV  | M,A    |                       |
| 125 | 36FD | 3A6637 | MAYBE | LDA  | LASTIN | GET EXIT CHARACTER    |
| 126 | 3700 | FEA0   |       | CPI  | ','    | NEXT LOCATION?        |
| 127 | 3702 | 2003   | \$    | BNE  | INS2   |                       |
| 128 | 3704 | 23     |       | INX  | H      | YES, BUMP POINTER     |
| 129 | 3705 | 18BA   | \$    | BRA  | INS1   | AND DO IT             |
| 130 | 3707 | FEAC   | INS2  | CPI  | ','    | BACK UP ONE LOCATION? |
| 131 | 3709 | C21436 |       | JNE  | RESUME | NO ABORT              |
| 132 | 370C | 2B     |       | DCX  | H      | YES, DO IT            |
| 133 | 370D | C3C136 |       | JMP  | INS1   |                       |
| 134 |      |        |       |      |        |                       |
| 135 |      |        |       |      |        |                       |
| 136 |      |        |       |      |        |                       |
| 137 | 3710 | 210000 |       |      |        |                       |
| 138 | 3713 | 54     |       |      |        |                       |
| 139 | 3714 | 44     |       |      |        |                       |
| 140 | 3715 | CD4637 |       |      |        |                       |
| 141 | 3718 | D8     |       |      |        |                       |
| 142 | 3719 | 29     |       |      |        |                       |
| 143 | 371A | 29     |       |      |        |                       |
| 144 | 371B | 29     |       |      |        |                       |
| 145 | 371C | 29     |       |      |        |                       |
| 146 | 371D | 5F     |       |      |        |                       |
| 147 | 371E | 19     |       |      |        |                       |
| 148 | 371F | 04     |       |      |        |                       |
| 149 | 3720 | 18F3   | \$    |      |        |                       |
| 150 |      |        |       |      |        |                       |
| 151 |      |        |       |      |        |                       |
| 152 |      |        |       |      |        |                       |
| 153 |      |        |       |      |        |                       |
| 154 | 3722 | 324437 |       |      |        |                       |
| 155 | 3725 | CD3937 |       |      |        |                       |
| 156 | 3728 | 324537 |       |      |        |                       |
| 157 | 372B | 3A4437 |       |      |        |                       |
| 158 | 372E | 1F     |       |      |        |                       |
| 159 | 372F | 1F     |       |      |        |                       |
| 160 | 3730 | 1F     |       |      |        |                       |
| 161 | 3731 | 1F     |       |      |        |                       |
| 162 | 3732 | CD3937 |       |      |        |                       |
| 163 | 3735 | 324437 |       |      |        |                       |
| 164 | 3738 | C9     |       |      |        |                       |
| 165 | 3739 | E60F   |       |      |        |                       |
| 166 | 373B | FE0A   |       |      |        |                       |
| 167 | 373D | 3802   |       |      |        |                       |
| 168 | 373F | C607   |       |      |        |                       |
| 169 | 3741 | C6B0   |       |      |        |                       |
| 170 | 3743 | C9     |       |      |        |                       |
| 171 | 3744 |        |       |      |        |                       |
| 172 |      |        |       |      |        |                       |
| 173 |      |        |       |      |        |                       |
| 174 |      |        |       |      |        |                       |
| 175 |      |        |       |      |        |                       |

```

176          * BIT RETURNS ON.
177          *
178 3746 CD6F36      FHCH      CALL      RKB***      GET TYPED CHARACTER
179 3749 326637      STA      LASTIN     SAVE IT
180 374C FEB0          CPI      :B0       LESS THAN :B0?
181 374E 3811          $        BLS      STC
182 3750 FEB8          CPI      :BA       GREATER THAN 9?
183 3752 380A          $        BLS      STRIP
184 3754 FEC1          CPI      'A'       IN A-F RANGE?
185 3756 3809          $        BLS      STC
186 3758 FEC7          CPI      'G'
187 375A 3005          $        BGE      STC
188 375C DE06          SBI      6         YES, BIAS DOWN
189 375E E60F          STRIP     ANI      :F       CLEAR UPPER 4 BITS AND CARRY
190 3760 C9           RET
191 3761 37           STC      STC
192 3762 3A6637      LDA      LASTIN     PICK UP INPUT CHAR.
193 3765 C9           RET              AND RETURN WITH CARRY ON
194 3766 00          LASTIN  ***
195 3767          SAVE      RES      2
196          *
197 3769 CD7637      PRT***   CALL      CRLF**
198 376C 7E          PRT2**   MOV      A,M
199 376D 23          INX      H
200 376E FEDC          CPI      '\ '
201 3770 C8          RZ
202 3771 CD7236      CALL      OUT***
203 3774 18F6          $      BRA      PRT2**
204          *
205 3776 3E8D          CRLF**   MVI      A,:8D
206 3778 CD7236      CALL      OUT***
207 377B 3E8A          MVI      A,:8A
208 377D CD7236      CALL      OUT***
209 3780 3EFF          MVI      A,:FF
210 3782 CD7236      CALL      OUT***
211 3785 C9          RET
212 3786 00          TEMP**   ***
213          *
214          * REGISTER EXAMINE/MODIFY
215          *
216 3787 CD6F36      AR       CALL      RKB***
217 378A 213339      LXI      H,A
218 378D FEA0          CPI      ' '
219 378F 280C          $      BEQ      ARI
220 3791 FEA7          $      CPI      :A7
221 3793 2022          $      BNE      ARTILT
222 3795 212B39      LXI      H,A'
223 3798 3EA0          MVI      A,' '
224 379A CD7236      CALL      OUT***
225 379D 22C137      ARI      SHLD     RADR
226 37A0 7E          MOV      A,M
227 37A1 CD2237      CALL      CNV2HX
228 37A4 CD7236      CALL      OUT***
229 37A7 3A4537      LDA      CN+1
230 37AA CD7236      CALL      OUT***
231 37AD 3EA0          MVI      A,' '
232 37AF CD7236      CALL      OUT***
233 37B2 CD1037      CALL      FHSTRG

```

|     |      |        |                        |      |         |
|-----|------|--------|------------------------|------|---------|
| 234 | 37B5 | FEAE   |                        | CPI  | '.'     |
| 235 | 37B7 | C25636 | ARTILT                 | JNE  | TILT    |
| 236 | 37BA | 78     |                        | MOV  | A,B     |
| 237 | 37BB | B7     |                        | ORA  | A       |
| 238 | 37BC | CA1436 |                        | JZ   | RESUME  |
| 239 | 37BF | 7D     |                        | MOV  | A,L     |
| 240 | 37C0 | 210000 | AR2                    | LXI  | H,0     |
| 241 |      |        | RADR                   | EQU  | AR2+1   |
| 242 | 37C3 | 77     |                        | MOV  | M,A     |
| 243 | 37C4 | C31436 |                        | JMP  | RESUME  |
| 244 |      |        | *                      |      |         |
| 245 |      |        | * EXAMINE/MODIFY FLAGS |      |         |
| 246 |      |        | *                      |      |         |
| 247 | 37C7 | CD6F36 | FR                     | CALL | RKB***  |
| 248 | 37CA | 213239 |                        | LXI  | H,F     |
| 249 | 37CD | FEA0   |                        | CPI  | '.'     |
| 250 | 37CF | 280D   | \$                     | BEQ  | FR1     |
| 251 | 37D1 | FEA7   |                        | CPI  | 'A7     |
| 252 | 37D3 | C25636 |                        | JNE  | TILT    |
| 253 | 37D6 | 212A39 |                        | LXI  | H,F'    |
| 254 | 37D9 | 3EA0   |                        | MVI  | A,''    |
| 255 | 37DB | CD7236 |                        | CALL | OUT***  |
| 256 | 37DE | 22C137 | FR1                    | SHLD | RADR    |
| 257 | 37E1 | 7E     |                        | MOV  | A,M     |
| 258 | 37E2 | 214D38 |                        | LXI  | H,DFRM1 |
| 259 | 37E5 | CD5E38 |                        | CALL | CNVTF   |
| 260 | 37E8 | 213C38 |                        | LXI  | H,FDSP  |
| 261 | 37EB | CD6937 |                        | CALL | PRT***  |
| 262 | 37EE | 3EA0   | FRA                    | MVI  | A,''    |
| 263 | 37F0 | CD7236 |                        | CALL | OUT***  |
| 264 | 37F3 | CD6F36 |                        | CALL | RKB***  |
| 265 | 37F6 | FEAE   |                        | CPI  | '.'     |
| 266 | 37F8 | CA1436 |                        | JEQ  | RESUME  |
| 267 | 37FB | 0680   |                        | MVI  | B,:80   |
| 268 | 37FD | FED3   |                        | CPI  | 'S'     |
| 269 | 37FF | 2823   | \$                     | BEQ  | FR2     |
| 270 | 3801 | CB38   | \$                     | LRB  |         |
| 271 | 3803 | FEDA   |                        | CPI  | 'Z'     |
| 272 | 3805 | 281D   | \$                     | BEQ  | FR2     |
| 273 | 3807 | CB38   | \$                     | LRB  |         |
| 274 | 3809 | CB38   | \$                     | LRB  |         |
| 275 | 380B | FEC1   |                        | CPI  | 'A'     |
| 276 | 380D | 2815   | \$                     | BEQ  | FR2     |
| 277 | 380F | CB38   | \$                     | LRB  |         |
| 278 | 3811 | CB38   | \$                     | LRB  |         |
| 279 | 3813 | FED6   |                        | CPI  | 'V'     |
| 280 | 3815 | 280D   | \$                     | BEQ  | FR2     |
| 281 | 3817 | CB38   | \$                     | LRB  |         |
| 282 | 3819 | FECE   |                        | CPI  | 'N'     |
| 283 | 381B | 2807   | \$                     | BEQ  | FR2     |
| 284 | 381D | CB38   | \$                     | LRB  |         |
| 285 | 381F | FEC3   |                        | CPI  | 'C'     |
| 286 | 3821 | C25636 | FTILT                  | JNE  | TILT    |
| 287 | 3824 | CD6F36 | FR2                    | CALL | RKB***  |
| 288 | 3827 | 2AC137 |                        | LHLD | RADR    |
| 289 | 382A | FEB1   |                        | CPI  | 'I'     |
| 290 | 382C | 280A   | \$                     | BEQ  | FR4     |
| 291 | 382E | FEB0   |                        | CPI  | '0'     |

PAGE 06 <<<< Z E D B U G M O N I T O R >>>>

```

292 3830 20EF      $      BNE  FTILT
293 3832 78        MOV   A,B
294 3833 2F        CMA
295 3834 A6        ANA   M
296 3835 77        FR3    MOV   M,A
297 3836 18B6      $      BRA   FRA
298 3838 78        FR4    MOV   A,B
299 3839 B6        ORA   M
300 383A 18F9      $      BRA   FR3
301 383C D3A0DAA0  FDSP    ASC   'S Z - A - V N C'
302 384B 8D8A      DATA  :8D, :8A
303 384D           DFRMI   RES   16, ' '
304 385D DC        DATA  '\ '
305
306               *
307               * CONVERT F REGISTER FOR DISPLAY IN
308               * AREA POINTED TO BY H&L
309               *
309 385E 47        CNVTF   MOV   B,A
310 385F 3E58      MVI   A, :58
311 3861 4F        MOV   C,A
312 3862 CB10     $      RBL
313 3864 17        RAL
314 3865 77        MOV   M,A      S
315 3866 23        INX   H
316 3867 23        INX   H
317 3868 79        MOV   A,C
318 3869 CB10     $      RBL
319 386B 17        RAL
320 386C 77        MOV   M,A      Z
321 386D 23        INX   H
322 386E 23        INX   H
323 386F 23        INX   H
324 3870 23        INX   H
325 3871 CB10     $      RBL
326 3873 CB10     $      RBL
327 3875 79        MOV   A,C
328 3876 17        RAL
329 3877 77        MOV   M,A      AC
330 3878 23        INX   H
331 3879 23        INX   H
332 387A 23        INX   H
333 387B 23        INX   H
334 387C CB10     $      RBL
335 387E CB10     $      RBL
336 3880 79        MOV   A,C
337 3881 17        RAL
338 3882 77        MOV   M,A      P/V
339 3883 23        INX   H
340 3884 23        INX   H
341 3885 CB10     $      RBL
342 3887 79        MOV   A,C
343 3888 17        RAL
344 3889 77        MOV   M,A      N
345 388A 23        INX   H
346 388B 23        INX   H
347 388C CB10     $      RBL
348 388E 79        MOV   A,C
349 388F 17        RAL

```



PAGE 07 <<<< Z E D B U G M O N I T O R >>>>

```

350 3890 77      MOV  M.A      C
351 3891 C9      RET
352
353      *
354      * EXAMINE/MODIFY REGISTER PAIRS
355 3892 213639  BCP      LXI      H.B
356 3895 180D    $      BRA      ARPEE
357 3897 213839  DEP      LXI      H.D
358 389A 1808    $      BRA      ARPEE
359 389C 213439  HLP      LXI      H.H
360 389F 1803    $      BRA      ARPEE
361 38A1 213E39  SP      LXI      H.STKP
362 38A4 CD6F36  ARPEE    CALL    RKB***
363 38A7 FEA0    CPI      /
364 38A9 280E    $      BEQ      RP2
365 38AB FEA7    CPI      :A7
366 38AD C25636  RP1     JNE      TILT
367 38B0 3EA0    MVI      A, /
368 38B2 CD7236  CALL    OUT***
369 38B5 11F8FF  LXI      D, -8
370 38B8 19      DAD      D
371 38B9 22EB38  RP2     SHLD    RPADR
372 38BC 23      INX      H
373 38BD 7E      MOV      A.M
374 38BE CD2237  CALL    CNV2HX
375 38C1 CD7236  CALL    OUT***
376 38C4 3A4537  LDA      CN+1
377 38C7 CD7236  CALL    OUT***
378 38CA 2B      DCX      H
379 38CB 7E      MOV      A.M
380 38CC CD2237  CALL    CNV2HX
381 38CF CD7236  CALL    OUT***
382 38D2 3A4537  LDA      CN+1
383 38D5 CD7236  CALL    OUT***
384 38D8 3EA0    MVI      A, /
385 38DA CD7236  CALL    OUT***
386 38DD CD1037  CALL    FHSTRG
387 38E0 FEA7    CPI      /
388 38E2 20C9    $      BNE      RP1
389 38E4 78      MOV      A.B
390 38E5 B7      ORA      A
391 38E6 CA1436  JZ      RESUME
392 38E9 EB      XCHG
393 38EA 210000  RP3     LXI      H.0      OPERAND FILLED IN ABOVE
394      RPADR    EQU      RP3+1
395 38ED 73      MOV      M.E
396 38EE 23      INX      H
397 38EF 72      MOV      M.D
398 38F0 C31436  JMP      RESUME
399 38F3 213A39  XREG     LXI      H.X
400 38F6 1803    $      BRA      YR2
401 38F8 213C39  YREG     LXI      H.Y
402 38FB CD6F36  YR2     CALL    RKB***
403 38FE FEA0    CPI      /
404 3900 20AB    $ RP4     BNE      RP1
405 3902 18B5    $      BRA      RP2
406
407      *
      * JUMP PROCESSOR

```

PAGE 08 <<<< Z E D B U G M O N I T O R >>>>

```

408
409 3904 CDD83D      *      JUMP      CALL      FPRM
410 3907 CA5636      JMI      JZ          TILT
411 390A FERE        CPI          ' '
412 390C 20F2      $      BNE      RP4
413 390E 222839      SHLD      JMP+I
414
415
416 3911 312A39      JM2      LXI      SP,F'
417 3914 F1          POP      PSW
418 3915 08          $      XAF
419 3916 E1          POP      H
420 3917 C1          POP      B
421 3918 D1          POP      D
422 3919 D9          $      EXX
423 391A F1          POP      PSW
424 391B E1          POP      H
425 391C C1          POP      B
426 391D D1          POP      D
427 391E DDE1      $      POP      X
428 3920 FDE1      $      POP      Y
429 3922 ED7B3E39 $      LSPD      STKP
430 3926 F3          E/I      DI
431 3927 C30000      JMP      JMP      0
432
433      * REGISTER STORAGE
434
435 392A 00          F'      ***
436 392B 00          A'      ***
437 392C 0000      H'      DBL      0
438 392E 0000      B'      DBL      0
439 3930 0000      D'      DBL      0
440 3932 00          F      ***
441 3933 00          A      ***
442 3934 0000      H      DBL      0
443 3936 0000      B      DBL      0
444 3938 0000      D      DBL      0
445 393A 0000      X      DBL      0
446 393C 0000      Y      DBL      0
447 393E 0000      STKP      DBL      0
448 3940 FFFF      BPA      DBL      -1
449 3942          BPI      RES      3
450 3945 0000      TSTACK      DBL      0
451 3947 0000      DIFF      DBL      0
452 3949 00          ILOC      ***
453 394A C3F839      RTINST      JMP      RTN
454 394D CECFA000      NPBMSG      ASC      'NO PREVIOUS BKPT\'
455 395E C2CBD004      BPRMSG      ASC      'BKPT REACHED AT '
456 396E          BPRLOC      RES      4
457 3972 DC          DATA      '\ '
458
459      * BREAKPOINT
460
461 3973 CDD83D      XEQ      CALL      FPRM
462 3976 288F      $ XEQ0      BZ          JMI
463 3978 FERE        CPI          ' '
464 397A 2807      $      BEQ      XEQ2
465 397C FERE        CPI          ' '

```

COMMON CODE TO LOAD REGISTER  
AND POINTERS.

INITIALIZE TO DISABLE

BKPT ADDRESS

NEW BREAKPOINT?  
YES  
USE THE OLD ONE?

PAGE 09 <<<< Z E D B U G M O N I T O R >>>>

```

466 397E C25636      XEQ1  JNE  TILT      NO, GARBAGE
467 3981 182B      $      BRA  XEQ4
468 3983 222839      XEQ2  SHLD  JMP+1    SET JUMP ADDRESS
469 3986 CDD83D      CALL  FPRM    GET BKPT ADDRESS
470 3989 28EB      $      BZ    XEQ0
471 398B FEAE      CPI      ' '      END IN PERIOD?
472 398D 20EF      $      BNE  XEQ1      NO, GARBAGE
473 398F 224039      XEQ3  SHLD  BPA      SAVE BKPT ADDRESS
474 3992 114239      LXI    D,BPI
475 3995 010300      LXI    B,3
476 3998 EDB0      $      MW+R      SAVE PROGRAM WORDS
477 399A ED5B4039 $      LDED  BPA
478 399E 214A39      LXI    H,RTINST
479 39A1 010300      LXI    B,3
480 39A4 EDB0      $      MW+R      MOVE IN BKPT RETURN INST.
481 39A6 3E01      MVI    A,1
482 39A8 324939      STA    ILOC
483 39AB C31139      JMP    JM2
484
485 39AE ED4B4039 $ XEQ4  LBCD  BPA      CONTINUE PREVIOUS BKPT
486 39B2 78      MOV    A,B      WAS THERE A PREVIOUS BKPT?
487 39B3 FEFF      CPI    :FF
488 39B5 200E      $      BNE  XEQ5
489 39B7 79      MOV    A,C
490 39B8 FEFF      CPI    :FF
491 39BA 2009      $      BNE  XEQ5
492 39BC 214D39      LXI    H,NPBMSG
493 39BF CD6937      CALL  PRT***
494 39C2 C31436      JMP    RESUME
495 39C5 ED4B4039 $ XEQ5  LBCD  BPA
496 39C9 ED432839 $      SBCD  JMP+1
497 39CD 18C0      $      BRA  XEQ3
498
499      *
500      * BREAKPOINT FAILURE RETURN
501      *
501 39CF CDE739      BPFAIL CALL  CLBKPT
502 39D2 21DB39      LXI    H,BPFMSG
503 39D5 CD6937      CALL  PRT***      PRINT FAIL MESSAGE
504 39D8 C31436      JMP    RESUME
505 39DB C2C8D0D4      BPFMSG ASC  'BKPT FAILED\
506
507      *
508      * CLEAR BREAKPOINT
509      *
509 39E7 ED5B4039 $ CLBKPT LDED  BPA
510 39EB 214239      LXI    H,BPI
511 39EE 010300      LXI    B,3
512 39F1 EDB0      $      MW+R
513 39F3 AF      ZAR
514 39F4 324939      STA    ILOC
515 39F7 C9      RET
516
517      *
518      * BREAKPOINT RETURN
519      *
519 39F8 F3      RTN      DI
520 39F9 ED734539 $      SSPD  TSTACK
521 39FD 313E39      LXI    SP,STKP
522 3A00 FDE5      $      PUSH  Y
523 3A02 DDE5      $      PUSH  X

```

```

524 3A04 D5      PUSH D
525 3A05 C5      PUSH B
526 3A06 E5      PUSH H
527 3A07 F5      PUSH PSW
528 3A08 D9      $ EXX
529 3A09 08      $ XAF
530 3A0A D5      PUSH D
531 3A0B C5      PUSH B
532 3A0C E5      PUSH H
533 3A0D F5      PUSH PSW
534 3A0E 310036   LXI SP,ZEDBUG
535 3A11 CDE739   CALL CLBKPT CLEAR THE BREAKPOINT
536 3A14 2A4539   LHL D TSTACK
537 3A17 ED5B3E39 $ LDED STKP
538 3A1B B7      RSC
539 3A1C ED52     $ DSC D FORM STACK DIFFERENCE
540 3A1E 224739   SHLD DIFF
541 3A21 2A4539   LHL D TSTACK
542 3A24 223E39   SHLD STKP
543 3A27 2A4039   LHL BPA
544 3A2A 116E39   LXI D,BPRLOC
545 3A2D CDB73B   CALL DCON
546 3A30 215E39   LXI H,BPRMSG
547 3A33 CD6937   CALL PRT***
548 3A36 C38A3A   JMP R2$
549
550 * SET ENABLE/DISABLE FLAG
551 *
552 3A39 CD6F36   IE CALL RKB***
553 3A3C 06F3     MVI B,:F3
554 3A3E FEC4     CPI 'D'
555 3A40 2804     $ BEQ IET
556 3A42 06FB     MVI B,:FB
557 3A44 FEC5     CPI 'E'
558 3A46 C25636   IET JNE TILT
559 3A49 CD6F36   CALL RKB***
560 3A4C FERE     CPI ' '
561 3A4E 20F6     $ BNE IET
562 3A50 78      IE2 MOV A,B
563 3A51 322639   STA E/I
564 3A54 C31436   JMP RESUME
565
566 *
567 * MODIFY/DISPLAY PSEUDO REGISTERS
568 *
569 3A57 CD6F36   R CALL RKB***
570 3A5A FERE     CPI ' '
571 3A5C CA8A3A   JEQ R2$
572 3A5F FEC1     CPI 'A'
573 3A61 CA8737   JEQ AR
574 3A64 FEC6     CPI 'F'
575 3A66 CAC737   JEQ FR
576 3A69 FEC2     CPI 'B'
577 3A6B CA9238   JEQ BCP
578 3A6E FEC4     CPI 'D'
579 3A70 CA9738   JEQ DEP
580 3A73 FEC8     CPI 'H'
581 3A75 CA9C38   JEQ HLP

```

|                 |         |          |                         |
|-----------------|---------|----------|-------------------------|
| 582 3A78 FED3   | CPI     | 'S'      |                         |
| 583 3A7A CAA138 | JEQ     | SP       |                         |
| 584 3A7D FED8   | CPI     | 'X'      |                         |
| 585 3A7F CAF338 | JEQ     | XREG     |                         |
| 586 3A82 FED9   | CPI     | 'Y'      |                         |
| 587 3A84 CAF838 | JEQ     | YREG     |                         |
| 588 3A87 C35636 | JMP     | TILT     |                         |
| 589 3A8A 3A3339 | R2# LDA | A        |                         |
| 590 3A8D CD2237 | CALL    | CNV2HX   | CONVERT A               |
| 591 3A90 32773B | STA     | AAF      |                         |
| 592 3A93 3A4537 | LDA     | CN+1     |                         |
| 593 3A96 32783B | STA     | AAF+1    |                         |
| 594 3A99 2A3639 | LHLD    | B        |                         |
| 595 3A9C 11673B | LXI     | D,ABCF   |                         |
| 596 3A9F CDB73B | CALL    | DCON     |                         |
| 597 3AA2 2A3839 | LHLD    | D        |                         |
| 598 3AA5 116C3B | LXI     | D,ADEF   |                         |
| 599 3AA8 CDB73B | CALL    | DCON     |                         |
| 600 3AAB 2A3439 | LHLD    | H        |                         |
| 601 3AAE 11713B | LXI     | D,AHLF   |                         |
| 602 3AB1 CDB73B | CALL    | DCON     |                         |
| 603 3AB4 3A3239 | LDA     | F        |                         |
| 604 3AB7 217D3B | LXI     | H,AF     |                         |
| 605 3ABA CD5E38 | CALL    | CNVTF    |                         |
| 606 3ABD 3EA0   | MVI     | A,' '    |                         |
| 607 3ABF 32663B | STA     | RFRAME   |                         |
| 608 3AC2 213F3B | LXI     | H,RLBLS  |                         |
| 609 3AC5 CD6937 | CALL    | PRT***   |                         |
| 610 3AC8 21663B | LXI     | H,RFRAME |                         |
| 611 3ACB CD6937 | CALL    | PRT***   |                         |
| 612 3ACE 3EA7   | MVI     | A,:A7    |                         |
| 613 3AD0 32663B | STA     | RFRAME   |                         |
| 614             |         |          | CONVERT ALTERNATE REG'S |
| 615 3AD3 3A2B39 | LDA     | A'       |                         |
| 616 3AD6 CD2237 | CALL    | CNV2HX   |                         |
| 617 3AD9 32773B | STA     | AAF      |                         |
| 618 3ADC 3A4537 | LDA     | CN+1     |                         |
| 619 3ADF 32783B | STA     | AAF+1    |                         |
| 620 3AE2 2A2E39 | LHLD    | B'       |                         |
| 621 3AE5 11673B | LXI     | D,ABCF   |                         |
| 622 3AE8 CDB73B | CALL    | DCON     |                         |
| 623 3AEB 2A3039 | LHLD    | D'       |                         |
| 624 3AEE 116C3B | LXI     | D,ADEF   |                         |
| 625 3AF1 CDB73B | CALL    | DCON     |                         |
| 626 3AF4 2A2C39 | LHLD    | H'       |                         |
| 627 3AF7 11713B | LXI     | D,AHLF   |                         |
| 628 3AFA CDB73B | CALL    | DCON     |                         |
| 629 3AFD 3A2A39 | LDA     | F'       |                         |
| 630 3B00 217D3B | LXI     | H,AF     |                         |
| 631 3B03 CD5E38 | CALL    | CNVTF    |                         |
| 632 3B06 21663B | LXI     | H,RFRAME |                         |
| 633 3B09 CD6937 | CALL    | PRT***   |                         |
| 634             |         |          | CONVERT X, Y, SP, DIFF  |
| 635 3B0C 2A3A39 | LHLD    | X        |                         |
| 636 3B0F 11A33B | LXI     | D,AXF    |                         |
| 637 3B12 CDB73B | CALL    | DCON     |                         |
| 638 3B15 2A3C39 | LHLD    | Y        |                         |
| 639 3B18 11A83B | LXI     | D,AYF    |                         |



```

640 3B1B CDB73B      CALL DCON
641 3B1E 2A3E39      LHL D STKP
642 3B21 11AD3B      LXI D,ASPF
643 3B24 CDB73B      CALL DCON
644 3B27 2A4739      LHL D DIFF
645 3B2A 11B23B      LXI D,ADFRM
646 3B2D CDB73B      CALL DCON
647 3B30 218D3B      LXI H,XLBLS
648 3B33 CD6937      CALL PRT***
649 3B36 21A23B      LXI H,XFRAME
650 3B39 CD6937      CALL PRT***
651 3B3C C31436      JMP RESUME
652
653 * PRINT AREA
654
655 3B3F A0A0C2C3      RLBS ASC ' BC '
656 3B45 A0C4C5A0      ASC ' DE '
657 3B4A A0C8CCA0      ASC ' HL '
658 3B50 C1A0A0A0      ASC ' A '
659 3B56 D3A0DAA0      ASC ' S Z - A - V N C '
660 3B66 A0            RFRAME DATA
661 3B67 ABCF          RES 5, '
662 3B6C ADEF          RES 5, '
663 3B71 AHF           RES 6, '
664 3B77 AAF           RES 6, '
665 3B7D AFF           RES 15, '
666 3B8C DC            DATA
667 3B8D A0A0D8A0      XLBS ASC ' X '
668 3B93 A0D9A0A0      ASC ' Y '
669 3B98 A0D3D0A0      ASC ' SP '
670 3B9D C4C9C6C6      ASC ' DIFF '
671 3BA2 A0            XFRAME DATA
672 3BA3 AXF           RES 5, '
673 3BA8 AYF           RES 5, '
674 3BAD ASPF          RES 5, '
675 3BB2 ADFRM         RES 4, '
676 3BB6 DC            DATA
677
678 * CONVERT THE NUMBER IN H AND L TO 4 CHAR'S
679 * BEGINNING AT THE ADDRESS IN D AND E.
680
681 3BB7 7C            DCON MOV A,H
682 3BB8 CD2237        CALL CNV2HX
683 3BBB 12            STAX D
684 3BBC 13            INX D
685 3BBD 3A4537        LDA CN+1
686 3BC0 12            STAX D
687 3BC1 13            INX D
688 3BC2 7D            MOV A,L
689 3BC3 CD2237        CALL CNV2HX
690 3BC6 12            STAX D
691 3BC7 13            INX D
692 3BC8 3A4537        LDA CN+1
693 3BCB 12            STAX D
694 3BCC C9            RET
695
696 * CONVERT AND PRINT THE WORD IN A
697

```

```

698 3BCD CD2237      PW      CALL   CNV2HX
699 3BD0 CD7236      CALL   OUT***
700 3BD3 3A4537      LDA     CN+1
701 3BD6 CD7236      CALL   OUT***
702 3BD9 C9          RET
703
704      * INPUT FROM DEVICE N
705      *
706 3BDA CDD83D      INPUT    CALL   FPRM
707 3BD0 281C      $      BZ     INTLT2
708 3BDF FEAE      CPI     ' '
709 3BE1 2018      $      BNE    INTLT2
710 3BE3 7D      MOV     A,L
711 3BE4 32EB3B      STA     INPT2+1
712 3BE7 CD7637      INPT1    CALL   CRLF**
713 3BEA DB00      INPT2    IN     0
714 3BEC CDCD3B      CALL   PW
715 3BEF CD6F36      CALL   RKB***
716 3BF2 FEAE      CPI     ' '
717 3BF4 28F1      $      BEQ    INPT1
718 3BF6 FEAE      INPT3    CPI     ' '
719 3BF8 CA1436      INRSM    JEQ    RESUME
720 3BFB C35636      INTLT2   JMP    TILT
721
722      * OUTPUT TO DEVICE N
723      *
724 3BFE CDD83D      OUTPUT    CALL   FPRM      GET DEVICE #
725 3C01 28F8      $      BZ     INTLT2
726 3C03 FEAE      CPI     ' '
727 3C05 20F4      $      BNE    INTLT2
728 3C07 7D      MOV     A,L
729 3C08 321B3C      STA     OUTPT2+1
730 3C0B CDD83D      OUTPT1    CALL   FPRM
731 3C0E 28E6      $      BZ     INPT3
732 3C10 FEAE      CPI     ' '
733 3C12 2804      $      BEQ    OTPT1A
734 3C14 FEAE      CPI     ' '
735 3C16 20E3      $      BNE    INTLT2
736 3C18 F5      OTPT1A    PUSH   PSW
737 3C19 7D      MOV     A,L
738 3C1A D300      OUTPT2    OUT    0
739 3C1C F1      POP     PSW
740 3C1D FEAE      CPI     ' '
741 3C1F 28D7      $      BEQ    INRSM
742 3C21 18E8      $      BRA     OUTPT1
743
744      *
745      *****
746      * OBJECT LOADER FOR Z80 RESIDENT ASSEMBLER *
747      *****
748      *
749 3C23 CD6F36      LOADER    CALL   RKB***
750 3C26 FEAE      CPI     ' '
751 3C28 C25636      JNE     TILT
752 3C2B 0600      MYI     B,0
753 3C2D CD6336      CALL   INRDR      INITIALIZE 6820
754 3C30 CD6936      CALL   RDRSTT    START READER
755 3C33 CD653E      LOAD1    CALL   RDRFM    GET TAPE FRAME
756 3C36 FEFF      CPI     :FF      RECORD MARK

```

```

756 3C38 2805      $      BEQ   LOAD2
757 3C3A 05        DCR   B
758 3C3B 20F6      $      BNZ   LOAD1
759 3C3D 185A      $      BRA   LOADWT
760 3C3F 57        LOAD2  MOV   D.A      START CHECKSUM
761 3C40 CD653E    CALL   RDRFM
762 3C43 6F        MOV   L.A      GET RECORD ORIGIN
763 3C44 CD653E    CALL   RDRFM
764 3C47 67        MOV   H.A      ORIGIN IN H&L
765 3C48 CD653E    CALL   RDRFM    GET RECORD TYPE
766 3C4B B7        ORA   A
767 3C4C 2017      $      BNZ   TYPE1
768 3C4E CD653E    CALL   RDRFM    GET RECORD LENGTH
769 3C51 47        MOV   B.A      SAVE IN B
770 3C52 CD653E    LOAD3  CALL   RDRFM
771 3C55 77        MOV   M.A
772 3C56 23        INX   H      READ AND STORE RECORD
773 3C57 10F9      $      DBNZ  LOAD3
774 3C59 5A        MOV   E.D      SAVE THE CHECKSUM
775 3C5A CD653E    CALL   RDRFM    GET CHECKSUM
776 3C5D AB        XRA   E      VERIFY
777 3C5E 201E      $      BNZ   TYPE10
778 3C60 060A      LOAD4  MVI   B.10   SET INTERRECORD GAP LENGTH
779 3C62 C3333C    JMP   LOAD1
780
781      * LOAD "RES" TYPE RECORD
782      *
783 3C65 CD653E    TYPE1  CALL   RDRFM
784 3C68 32B83C    STA   FLFLAG    FILL FLAG
785 3C6B CD653E    CALL   RDRFM
786 3C6E 32B93C    STA   FLCHR    FILL CHARACTER
787 3C71 CD653E    CALL   RDRFM
788 3C74 4F        MOV   C.A      GET "RES" LENGTH
789 3C75 CD653E    CALL   RDRFM
790 3C78 47        MOV   B.A
791 3C79 5A        MOV   E.D      SAVE THE CHECKSUM
792 3C7A CD653E    CALL   RDRFM    GET CHECKSUM
793 3C7D AB        XRA   E      VERIFY...
794 3C7E 201F      $ TYPE10 BNZ   CKSMER  CHECKSUM ERROR
795 3C80 3AB83C    LDA   FLFLAG    WAS THERE A FILL CHARACTER?
796 3C83 B7        ORA   A
797 3C84 28DA      $      BZ    LOAD4
798 3C86 3AB93C    LDA   FLCHR    YES, GET IT
799 3C89 57        MOV   D.A
800 3C8A CD6C36    CALL   RDRSTP  JUST IN CASE...
801 3C8D 72        TYPE1A MOV   M.D
802 3C8E 23        INX   H
803 3C8F 0B        DCX   B
804 3C90 78        MOV   A.B
805 3C91 B1        ORA   C
806 3C92 20F9      $      BNZ   TYPE1A
807 3C94 CD6936    CALL   RDRSTT  RESTART READER
808 3C97 18C7      $      BRA   LOAD4
809
810 3C99 CD6C36    LOADWT  CALL   RDRSTP
811 3C9C C31436    JMP   RESUME
812
813      * CHECKSUM ERROR PROCEDURE

```

```

814                                     *
815 3C9F CD6C36      CKSMER  CALL  RDRSTP
816 3CA2 3EC3        MVI    A,'C'
817 3CA4 CD7236      CALL  OUT***
818 3CA7 3ECB        MVI    A,'K'
819 3CA9 CD7236      CALL  OUT***
820 3CAC 3ED3        MVI    A,'S'
821 3CAE CD7236      CALL  OUT***
822 3CB1 3ECD        MVI    A,'M'
823 3CB3 CD7236      CALL  OUT***
824 3CB6 18E1        $      BRA   LOADWT
825                                     *
826 3CB8 00          FLFLAG ***
827 3CB9 00          FLCHR  ***
828                                     *
829                                     * DUMP MEMORY BLOCK IN RELOADABLE FORMAT
830                                     *
831 3CBA CDD83D      DUMP    CALL  FPRM      GET FIRST ADDRESS
832 3CBD CA5636      DUMP0   JZ     TILT     NOTHING TYPED
833 3CC0 FE80        CPI     ' '          MUST BE BLANK
834 3CC2 C25636      JNE     TILT
835 3CC5 22593D      SHLD    DTEMP      SAVE START ADDRESS
836 3CC8 CDD83D      CALL    FPRM      GET SECOND ADDRESS
837 3CCB 28F0        $      BZ     DUMP0
838 3CCD FEAE        CPI     ' '          MUST BE PERIOD
839 3CCF C25636      DUMP0A  JNE     TILT
840 3CD2 ED5B593D    $      LDED    DTEMP
841 3CD6 37          STC
842 3CD7 EB          XCHG
843 3CD8 ED52        $      DSC     D      FORM INCLUSIVE NEGATIVE DIFF
844 3CDA 7C          MOV     A,H      GET HIGH ORDER
845 3CDB B7          ORA     A
846 3CDC F25636      JP      TILT      1ST ADDRESS > 2ND ADDRESS
847 3CDF CD7E36      CALL    SB       WAIT FOR SPACE
848 3CE2 063C        MVI    B,60     GET LEADER LENGTH
849 3CE4 AF          ZAR
850 3CE5 57          MOV     D,A
851 3CE6 CD5E3D      DUMP1   CALL    PUNCH   PUNCH LEADER
852 3CE9 10FB        $      DBNZ   DUMP1
853 3CEB DD2A593D    $      LXD     DTEMP   START ADDRESS IN X
854 3CEF 24          DUMP2   INR     H      LAST PAGE?
855 3CF0 2829        $      BZ     LRCORD
856 3CF2 CD423D      CALL    PROLOG   PUNCH R.M. & ORIGIN
857 3CF5 AF          ZAR
858 3CF6 CD5E3D      CALL    PUNCH   PUNCH RECORD LENGTH
859 3CF9 DD7E00      $      LAX     0      GET FRAME FROM MEMORY
860 3CFC CD5E3D      CALL    PUNCH
861 3CFF DD23        $      INX     X
862 3D01 CD7B36      CALL    DTCTCC
863 3D04 DA1436      JC      RESUME
864 3D07 10F0        $      DBNZ   DUMP3   PUNCH RECORD
865 3D09 7A          MOV     A,D      GET CHECKSUM
866 3D0A CD5E3D      CALL    PUNCH   PUNCH IT
867 3D0D AF          ZAR
868 3D0E CD5E3D      CALL    PUNCH   PUNCH INTERRECORD GAP
869 3D11 AF          ZAR
870 3D12 CD5E3D      CALL    PUNCH
871 3D15 DD22593D    $      SXD     DTEMP   SAVE ADDRESS

```

PAGE 16 <<<< Z E D B U G M O N I T O R >>>>

```

872 3D19 18D4      $      BRA      DUMP2
873 3D1B CD423D      LRCORD  CALL    PROLOG      PUNCH LAST RECORD PROLOGUE
874 3D1E 7D          MOV      A,L          GET LAST RECORD LENGTH
875 3D1F ED44      $      NEG
876 3D21 47          MOV      B,A
877 3D22 CD5E3D      CALL    PUNCH      PUNCH RECORD LENGTH
878 3D25 DD7E00      $ DUMP4  LAX      0          GET FRAME
879 3D28 CD5E3D      CALL    PUNCH
880 3D2B DD23      $      INX      X
881 3D2D 10F6      $      DBNZ    DUMP4
882 3D2F 7A          MOV      A,D
883 3D30 CD5E3D      CALL    PUNCH      PUNCH CHECKSUM
884 3D33 AF          ZAR
885 3D34 57          MOV      D,A
886 3D35 0646      MVI      B,70      TRAILER LENGTH
887 3D37 CD5E3D      DUMP5  CALL    PUNCH
888 3D3A 10FB      $      DBNZ    DUMP5
889 3D3C CD7E36      CALL    SB          WAIT FOR SPACE
890 3D3F C31436      JMP      RESUME      RETURN TO MONITOR
891
892      * PUNCH R.M., ORIGIN AND TYPE
893      *
894 3D42 3EFF      PROLOG  MVI      A,:FF      RUBOUT FOR RECORD MARK
895 3D44 CD5E3D      CALL    PUNCH
896 3D47 57          MOV      D,A          INITIALIZE CHECKSUM
897 3D48 3A593D      LDA      DTEMP      LOW ORIGIN ADDRESS
898 3D4B CD5E3D      CALL    PUNCH
899 3D4E 3A5A3D      LDA      DTEMP+1
900 3D51 CD5E3D      CALL    PUNCH
901 3D54 AF          ZAR
902 3D55 CD5E3D      CALL    PUNCH      PUNCH RECORD TYPE
903 3D58 C9          RET
904 3D59      DTEMP  RES      5,0
905      *
906      * PUNCH AND KEEP CHECKSUM
907      *
908 3D5E F5          PUNCH  PUSH    PSW
909 3D5F 82          ADD      D
910 3D60 57          MOV      D,A
911 3D61 F1          POP      PSW
912 3D62 CD7836      CALL    PCHCHR
913 3D65 C9          RET
914
915      * PRINT MEMORY BLOCK
916      *
917 3D66 22593D      PBLK   SHLD   DTEMP
918 3D69 CDD83D      CALL    FPRM
919 3D6C CA5636      JZ      TILT
920 3D6F FEAE      CPI      ' '          MUST END IN PERIOD
921 3D71 C25636      JNE    TILT
922 3D74 ED58593D $      LDED   DTEMP
923 3D78 B7          RSC
924 3D79 ED52      $      DSC      D
925 3D7B 23          INX      H
926 3D7C 7C          MOV      A,H
927 3D7D B7          ORA      A
928 3D7E FA5636      JM      TILT
929 3D81 B5          ORA      L

```



```

<<<<< Z E D B U G   M O N I T O R >>>>

```

```

930 3D82 CAFE636                JZ      TILT
931 3D85 DD2A593D $             LXD      DTEMP
932 3D89 CD7637                PBI     CALL    CRLF**
933 3D8C 3A5A3D                LDA      DTEMP+1
934 3D8F CD2237                CALL    CNV2HX
935 3D92 CD7236                CALL    OUT***
936 3D95 3A4537                LDA      CN+1
937 3D98 CD7236                CALL    OUT***
938 3D9B 3A593D                LDA      DTEMP
939 3D9E CD2237                CALL    CNV2HX
940 3DA1 CD7236                CALL    OUT***
941 3DA4 3A4537                LDA      CN+1
942 3DA7 CD7236                CALL    OUT***
943 3DA9 3EA0                  MVI     A, '/'
944 3DAC CD7236                CALL    OUT***
945 3DAF 0610                  MVI     B, 16          16 #'S PER LINE
946 3DB1 3EA0                PBI     MVI     A, '/'
947 3DB3 CD7236                CALL    OUT***
948 3DB6 DD7E00                $      LAX      0          GET MEMORY WORD
949 3DB9 DD23                  $      INX      X
950 3DBB CD2237                CALL    CNV2HX
951 3DBE CD7236                CALL    OUT***
952 3DC1 3A4537                LDA      CN+1
953 3DC4 CD7236                CALL    OUT***
954 3DC7 28                    DCX      H
955 3DC8 7C                    MOV     A, H
956 3DC9 B5                    ORA      L
957 3DCA CA1436                JZ      RESUME
958 3DCD 10E2                  $      DBNZ    PB2
959 3DCF DD22593D $           SXD      DTEMP
960 3DD3 CD7B36                CALL    DTCTCC
961 3DD6 DA1436                JC      RESUME
962 3DD9 18AE                  $      BRA     PBI
963                               *
964                               *  FETCH PARAMETER FROM KEYBOARD
965                               *
966 3DDB CD1037                FPRM     CALL    FHSTRG    GET HEX NUMBER
967 3DDE 32E73D                STA      FTEMP
968 3DE1 AF                    ZAR
969 3DE2 80                    ADD     B          SET Z BIT IF NOTHING TYPED
970 3DE3 3AE73D                LDA      FTEMP
971 3DE6 C9                    RET
972 3DE7 00                FTEMP   ***
973                               *
974                               *  FILL MEMORY BLOCK
975                               *
976 3DE8 CD273E                FILL     CALL    GETPRM
977 3DEB 54                    MOV     D, H
978 3DEC 5D                    MOV     E, L
979 3DED 13                    INX      D
980 3DEE 3A5D3D                LDA      DTEMP+4
981 3DF1 77                    MOV     M, A
982 3DF2 EDB0                $      MW+R
983 3DF4 C31436                JMP      RESUME
984                               *
985                               *  SEARCH MEMORY BLOCK FOR VALUE
986                               *
987 3DF7 CD273E                SEARCH  CALL    GETPRM

```

```

988 3DFA 03          INX  B          FORM INCLUSIVE DIFFERENCE
989 3DFB 3A5D3D      SR1  LDA  DTEMP+4
990 3DFE BE          CMP  M
991 3DFF 280F        $    BEQ  SRF
992 3E01 CD7B36      SR2  CALL DTCTCC
993 3E04 DA1436      SR3  JC   RESUME
994 3E07 23          INX  H
995 3E08 0B          DCX  B
996 3E09 78          MOV  A,B
997 3E0A B1          ORA  C
998 3E0B 20EE        $    BNZ  SR1
999 3E0D C31436      JMP  RESUME
000 3E10 C5          SRF  PUSH B
001 3E11 E5          PUSH H
002 3E12 11223E      LXI  D,FADRS
003 3E15 CDB73B      CALL DCON
004 3E18 21223E      LXI  H,FADRS
005 3E1B CD6937      CALL PRT***
006 3E1E E1          POP  H
007 3E1F C1          POP  B
008 3E20 18DF        $    BRA  SR2
009 3E22 B0B0B0B0    FADRS ASC '0000\
010                  *
011                  * GET PARAMETERS FOR FILL AND SEARCH
012                  *
013 3E27 CDD83D      GETPRM CALL FPRM
014 3E2A CA5636      GT2  JZ   TILT
015 3E2D FEA0        CPI  /
016 3E2F C25636      GT3  JNE  TILT
017 3E32 22593D      SHLD DTEMP
018 3E35 CDD83D      CALL FPRM
019 3E38 28F0        $    BZ   GT2
020 3E3A FEA0        CPI  /
021 3E3C 20F1        $    BNE  GT3
022 3E3E 225B3D      SHLD DTEMP+2
023 3E41 CDD83D      CALL FPRM
024 3E44 28E4        $    BZ   GT2
025 3E46 FEAE        CPI  /
026 3E48 20E5        $    BNE  GT3
027 3E4A 7D          MOV  A,L
028 3E4B 325D3D      STA  DTEMP+4
029 3E4E ED5B593D    $    LDED DTEMP
030 3E52 2A5B3D      LHL DTEMP+2
031 3E55 B7          RSC
032 3E56 ED52        $    DSC  D
033 3E58 DA5636      JC   TILT
034 3E5B 7C          MOV  A,H
035 3E5C B5          ORA  L
036 3E5D 28CB        $    BZ   GT2
037 3E5F 44          MOV  B,H      MOVE COUNT TO B-C
038 3E60 4D          MOV  C,L
039 3E61 2A593D      LHL DTEMP
040 3E64 C9          RET
041                  *
042                  * GET TAPE FRAME, KEEP CHECKSUM IN D
043                  *
044 3E65 CD6636      RDFRM CALL GTFRM
045 3E68 F5          PUSH PSW

```

PAGE 19 <<<< Z E D B U G M O N I T O R >>>>

|             |     |     |
|-------------|-----|-----|
| 046 3E69 82 | ADD | D   |
| 047 3E6A 57 | MOV | D,A |
| 048 3E6B F1 | POP | PSW |
| 049 3E6C C9 | RET |     |
| 050 3E6D    | END |     |

\*\*\*\*\*  
 \* S Y M B O L T A B L E \*  
 \*\*\*\*\*

|        |      |        |      |        |      |        |      |
|--------|------|--------|------|--------|------|--------|------|
| A      | 3933 | A'     | 392B | AAF    | 3B77 | ABCF   | 3B67 |
| ADEF   | 3B6C | ADFRM  | 3BB2 | AFF    | 3B7D | AHLF   | 3B71 |
| AR     | 3787 | ARI    | 379D | AR2    | 37C0 | ARPEE  | 3BA4 |
| ARTILT | 37B7 | ASPF   | 3BAD | AXF    | 3BA3 | AYF    | 3BA8 |
| B      | 3936 | B'     | 392E | BCP    | 3892 | BPA    | 3940 |
| BPFAIL | 39CF | BPFMSG | 39DB | BPI    | 3942 | BPRLOC | 396E |
| BPRMSG | 395E | CKSMER | 3C9F | CLBKPT | 39E7 | CN     | 3744 |
| CNV2HX | 3722 | CNVTF  | 385E | CRLF** | 3776 | CSTACK | 3681 |
| CSTK2  | 368F | CSTK3  | 36AE | D      | 3938 | D'     | 3930 |
| DCON   | 3BB7 | DEP    | 3897 | DFRM1  | 384D | DIFF   | 3947 |
| DTCTCC | 367B | DEMP   | 3D59 | DUMP   | 3CBA | DUMPO  | 3C8D |
| DUMPOA | 3CCF | DUMPI  | 3CE6 | DUMP2  | 3CEF | DUMP3  | 3CF9 |
| DUMP4  | 3D25 | DUMP5  | 3D37 | E/I    | 3926 | F      | 3932 |
| F'     | 392A | FADRS  | 3E22 | FDSP   | 383C | FHI    | 3715 |
| FHCH   | 3746 | FHSTRG | 3710 | FILL   | 3DE8 | FLCHR  | 3CB9 |
| FLFLAG | 3CB8 | FPRM   | 3DDB | FR     | 37C7 | FRI    | 37DE |
| FR2    | 3824 | FR3    | 3835 | FR4    | 3838 | FRA    | 37EE |
| FTEMP  | 3DE7 | FTILT  | 3821 | GETPRM | 3E27 | GT2    | 3E2A |
| GT3    | 3E2F | GTFRM  | 3666 | H      | 3934 | H'     | 392C |
| HLP    | 389C | IE     | 3A39 | IE2    | 3A50 | IET    | 3A46 |
| ILOC   | 3949 | INPT1  | 3BE7 | INPT2  | 3BEA | INPT3  | 3BF6 |
| INPUT  | 3BDA | INRDR  | 3663 | INRSM  | 3BF8 | INS    | 36B2 |
| INS1   | 36C1 | INS2   | 3707 | INS3   | 36E8 | INS3A  | 36F8 |
| INTLT2 | 3BFB | INTMNL | 3660 | JM1    | 3907 | JM2    | 3911 |
| JMP    | 3927 | JUMP   | 3904 | LASTIN | 3766 | LOAD1  | 3C33 |
| LOAD2  | 3C3F | LOAD3  | 3C52 | LOAD4  | 3C60 | LOADER | 3C23 |
| LOADWT | 3C99 | LRCORD | 3D1B | MAYBE  | 36FD | NPBMSG | 394D |
| OTPT1A | 3C18 | OUT*** | 3672 | OUTPT1 | 3C0B | OUTPT2 | 3C1A |
| OUTPUT | 3BFE | PBI    | 3D89 | PB2    | 3DB1 | PBLK   | 3D66 |
| PCHCHR | 3678 | PDVCIN | 3675 | PROLOG | 3D42 | PRT*** | 3769 |
| PRT2** | 376C | PUNCH  | 3D5E | PW     | 3BCD | R      | 3A57 |
| R2\$   | 3A8A | RADR   | 37C1 | RDFRM  | 3E65 | RDRSTP | 366C |
| RDRSTT | 3669 | RESUME | 3614 | RFRAME | 3B66 | RKB*** | 366F |
| RLBL5  | 3B3F | RPI    | 38AD | RP2    | 38B9 | RP3    | 38EA |
| RP4    | 3900 | RPADR  | 38EB | RTINST | 394A | RTN    | 39F8 |
| SAVE   | 3767 | SB     | 367E | SEARCH | 3DF7 | SP     | 38A1 |
| SR1    | 3DFB | SR2    | 3E01 | SR3    | 3E04 | SRF    | 3E10 |
| STC    | 3761 | STKP   | 393E | STRIP  | 375E | TEMP** | 3786 |
| TILT   | 3656 | TSTACK | 3945 | TYPE1  | 3C65 | TYPE10 | 3C7E |
| TYPE1A | 3C8D | X      | 393A | XEQ    | 3973 | XEQ0   | 3976 |
| XEQ1   | 397E | XEQ2   | 3983 | XEQ3   | 398F | XEQ4   | 39AE |
| XEQ5   | 39C5 | XFRAME | 3BA2 | XLBL5  | 3B8D | XREG   | 38F3 |
| Y      | 393C | YR2    | 38FB | YREG   | 38F8 | ZEDBUG | 3600 |
| ZH2    | 3741 | ZHEX   | 3739 |        |      |        |      |

## APPENDIX D / CROSS INDEX OF ASSEMBLER MNEMONICS WITH ZILOG MNEMONICS

This appendix contains a table of correspondences of the mnemonics used by the assembler in appendix B with those described in *Z80-Assembly Language Programming Manual*, 3.ODS REL 2.1 (Zilog, Inc. 1977), and a short description of the function of each instruction and its effect on the flags. The terms used in instruction operands have the following meanings:

- aa —a memory address
- b —a bit, numbered 0 - 7 from right to left
- d —an 8 bit displacement in the range +127 to -128
- n —an 8 bit constant
- nn —a 16 bit constant
- r —a single register, value defined below
- t —a 3 bit constant
- la —low 8 bits of 16 bit address
- ha —high 8 bits of 16 bit address
- ln —low 8 bits of 16 bit constant
- hn —high 8 bits of 16 bit constant

The Assembly Scheme column contains the description of the successive words of the instruction, separated by commas. The symbol V has the meaning Inclusive OR. Numbers in this column are all hexadecimal. The assembly scheme for **MOV r<sub>1</sub>r<sub>2</sub>**, for example, is 40V8r<sub>1</sub>Vr<sub>2</sub>, meaning the logical sum of 40<sub>16</sub>, 8 times the value of r<sub>1</sub>, and the value of r<sub>2</sub>. The register values are:

B — 0  
 C — 1  
 D — 2  
 E — 3  
 H — 4  
 L — 5  
 A — 7

The symbols in the flag setting column, labeled **SZAVNC**, have the following meanings:

? — flag value unknown after this instruction  
 — — flag value unchanged after this instruction  
 0 — flag cleared to zero by this instruction  
 1 — flag set to one by this instruction  
 x — flag set to reflect outcome of instruction

Blank in this column means that all flags are unchanged by the instruction.

The column headed Zilog name/page gives the equivalent mnemonic in *Z80-Assembly Language Programming Manual 3.0 DS REL 2.1*, and the page in the Zilog manual on which the description of this instruction is to be found.

In all columns a parenthesized quantity means the contents of the memory address pointed to by that quantity, i.e., (HL) means the contents of the memory location whose address is in HL.



## MNEMONIC CROSS INDEX

| mnemonic |     | assembly/scheme | S Z A V N C | Zilog name | /        | page |
|----------|-----|-----------------|-------------|------------|----------|------|
| ACI      | n   | CE,n            | XXXXX0 X    | ADC        | A,S      | 108  |
| ADC      | M   | 8E              | XXXXX0 X    | ADC        | A,(HL)   | 108  |
| ADC      | r   | 88Vr            | XXXXX0 X    | ADC        | A,S      | 108  |
| ADCX     | d   | DD,8E,d         | XXXXX0 X    | ADC        | A,(IX+d) | 108  |
| ADCY     | d   | FD,8E,d         | XXXXX0 X    | ADC        | A,(IY+d) | 108  |
| ADD      | M   | 86              | XXXXX0 X    | ADD        | A,(HL)   | 103  |
| ADD      | r   | 80Vr            | XXXXX0 X    | ADD        | A,r      | 100  |
| ADDX     | d   | DD,86,d         | XXXXX0 X    | ADD        | A,(IX+d) | 104  |
| ADDY     | d   | FD,86,d         | XXXXX0 X    | ADD        | A,(IY+d) | 106  |
| ADI      | n   | C6,n            | XXXXX0 X    | ADD        | A,n      | 102  |
| ANA      | M   | A6              | XX1 X0 0    | AND        | (HL)     | 114  |
| ANA      | r   | 50Vr            | XX1 X0 0    | AND        | r        | 114  |
| ANI      | n   | E6,n            | XX1 X0 0    | AND        | n        | 114  |
| ANX      | d   | DD,A6,d         | XX1 X0 0    | AND        | (IX+d)   | 114  |
| ANY      | d   | FD,A6,d         | XX1 X0 0    | AND        | (IY+d)   | 114  |
| ARM      |     | CB,2E           | XX0 X0 X    | SRA        | (HL)     | 192  |
| ARMX     | d   | DD,CB,D,2E      | XX0 X0 X    | SRA        | (IX+d)   | 192  |
| ARMY     | d   | FD,CB,D,2E      | XX0 X0 X    | SRA        | (IY+d)   | 192  |
| ARr      |     | CB,28Vr         | XX0 X0 X    | SRA        | r        | 192  |
| BC       | aa  | 38,aa-PC        |             | JR         | C,e      | 224  |
| BCM      | b   | CB,46V8b        |             | RES        | b,(HL)   | 217  |
| BCMX     | d,b | DD,CB,d,46V8b   |             | RES        | b,(IX+d) | 217  |
| BCMY     | d,b | FD,BC,d,46V8b   |             | RES        | b,(IY+d) | 217  |
| BCr      | b   | CB,80V8bVr      |             | RES        | b,r      | 217  |
| BEQ      | aa  | 28,aa-PC        |             | JR         | Z,e      | 228  |
| BGE      | aa  | 30,aa-PC        |             | JR         | NC,e     | 226  |
| BLS      | aa  | 38,aa-PC        |             | JR         | C,e      | 224  |
| BNE      | aa  | 20,aa-PC        |             | JR         | NC,e     | 230  |
| BNZ      | aa  | 20,aa-PC        |             | JR         | NZ,e     | 230  |
| BRA      | aa  | 18,aa-PC        |             | JR         | e        | 223  |
| BSM      | b   | CB,C6V8b        |             | SET        | b,(HL)   | 212  |
| BSMX     | d,b | DD,CB,d,C6V8b   |             | SET        | b,(IX+d) | 213  |
| BSMY     | d,b | FD,CB,d,C6V8b   |             | SET        | b,(IY+d) | 215  |
| BSr      | b   | CB,COV8bVr      |             | SET        | b,r      | 211  |
| BTM      | b   | CB,46V8b        | ? X1 ? 0 -  | BIT        | b,(HL)   | 205  |
| BTMX     | d,b | DD,CB,d,46V8b   | ? X1 ? 0 -  | BIT        | b,(IX+d) | 207  |
| BTMY     | d,b | FD,CB,d,46V8b   | ? X1 ? 0 -  | BIT        | b,(IY+d) | 209  |

**instruction description**

Add immediate data n and previous carry to A  
Add (HL) and previous carry to A  
Add r and previous carry to A  
Add (X+d) and previous carry to A  
Add (Y+d) and previous carry to A  
Add (HL) to A  
Add register r to A, sum in A  
Add (X+d) to A  
Add (Y+d) to A  
Add immediate data n to A  
AND (HL) with A  
AND register r with A  
AND immediate data n with A  
AND (X+d) with A  
AND (Y+d) with A  
Arithmetic right shift (HL)  
Arithmetic right shift (X+d)  
Arithmetic right shift (Y+d)  
Arithmetic right shift register r  
Branch if carry to address aa  
Clear bit b of (HL)  
Clear bit b of (X+d)  
Clear bit b of (Y+d)  
Clear bit b of register r  
Branch if equal to address aa  
Branch if greater than or equal to address aa (unsigned)  
Branch if less to address aa (unsigned)  
Branch if not equal to address aa  
Branch if not zero to address aa  
Branch to address aa unconditionally  
Set bit b of (HL)  
Set bit b of (X+d)  
Set bit b of (Y+d)  
Set bit b of register r  
Test bit b of (HL)  
Test bit b of (X+d)  
Test bit b of (Y+d)

| mnemonic |    | assembly/scheme | S Z A V N C | Zilog name | /      | page |
|----------|----|-----------------|-------------|------------|--------|------|
| BTr      | b  | CB,40V8bVr      | ? X 1 ? 0 - | BIT        | b,r    | 203  |
| BZ       | aa | 28,aa-PC        |             | JR         | Z,e    | 228  |
| CALL     | aa | CD,la,ha        |             | CALL       | nn     | 238  |
| CC       | aa | DC,la,ha        |             | CALL       | cc,nn  | 240  |
| CEQ      | aa | CC,la,ha        |             | CALL       | cc,nn  | 240  |
| CGE      | aa | D4,la,ha        |             | CALL       | cc,nn  | 240  |
| CLS      | aa | DC,la,ha        |             | CALL       | cc,nn  | 240  |
| CM       |    | FC,la,ha        |             | CALL       | cc,nn  | 240  |
| CMA      |    | 2F              | - - 1 - 1 - | CPL        |        | 134  |
| CMC      |    | 3F              | - - - - 0 X | CCF        |        | 137  |
| CMP      | M  | BE              | XXXXX1 X    | CP         | (HL)   | 120  |
| CMP      | r  | B8Vr            | XXXXX1 X    | CP         | s      | 120  |
| CMPX     | d  | DD,BE,d         | XXXXX1 X    | CP         | (IX+d) | 120  |
| CMPY     | d  | FD,BE,d         | XXXXX1 X    | CP         | (IY+d) | 120  |
| CNC      | aa | D4,la,ha        |             | CALL       | cc,nn  | 240  |
| CNE      |    | C4,la,ha        |             | CALL       | cc,nn  | 240  |
| CNV      |    | E4,la,ha        |             | CALL       | cc,nn  | 240  |
| CNZ      |    | C4,la,ha        |             | CALL       | cc,nn  | 240  |
| CP       |    | F4,la,ha        |             | CALL       | cc,nn  | 240  |
| CPE      |    | EC,la,ha        |             | CALL       | cc,nn  | 240  |
| CPI      | n  | FE,n            | XXXXX1 X    | CP         | n      | 120  |
| CPO      |    | E4,la,ha        |             | CALL       | cc,nn  | 240  |
| CV       |    | EC,la,ha        |             | CALL       | cc,nn  | 240  |
| CZ       | aa | CC,la,ha        |             | CALL       | cc,nn  | 240  |
| CP-      |    | ED,A9           | XXXXX1 -    | CPD        |        | 95   |
| CP-R     |    | ED,B9           | XXXXX1 -    | CPDR       |        | 97   |
| CP+      |    | ED,A1           | XXXXX1 -    | CPI        |        | 91   |
| CP+R     |    | ED,B2           | XXXXX1 -    | CPIR       |        | 93   |
| C<       | aa | FC,la,ha        |             | CALL       | cc,nn  | 240  |
| C>=      | aa | F4,la,ha        |             | CALL       | cc,nn  | 240  |
| DAA      |    | 27              | XXXXX- X    | DAA        |        | 132  |
| DAC      | B  | ED,4A           | XXXXX0 X    | ADC        | HL,ss  | 149  |
| DAC      | D  | ED,5A           | XXXXX0 X    | ADC        | HL,ss  | 149  |
| DAC      | H  | ED,6A           | XXXXX0 X    | ADC        | HL,ss  | 149  |
| DAC      | SP | ED,7A           | XXXXX0 X    | ADC        | HL,ss  | 149  |

**instruction description**

Test bit b of register r  
Branch if zero to address aa  
Unconditional call to address aa  
Call address aa if carry  
Call address aa if equal  
Call address aa if greater than or equal (unsigned)  
Call address aa if less (unsigned)  
Call address aa if minus  
Logical complement A (one's complement)  
Logical complement carry bit  
Compare A to (HL)  
Compare A to register r  
Compare A to (X+d)  
Compare A to (Y+d)  
Call address aa if no carry  
Call address aa if not equal  
Call address aa if no overflow  
Call address aa if not zero  
Call address aa if plus  
Call address aa if parity even  
Compare A to immediate data n  
Call address aa if parity odd  
Call address aa if overflow  
Call address aa if zero  
Compare A to (HL), decrement HL and BC, set flags  
Compare A to (HL), decrement HL and BC, stop if BC = 0 or  
equal compare  
Compare A to (HL), increment HL, decrement BC, set flags  
Compare A to (HL), increment HL, decrement BC, stop if BC = 0 or  
equal compare  
Call address aa if less (signed)  
Call address aa if greater than or equal (signed)  
Decimal adjust A register after BCD addition or subtraction  
Add BC and previous carry to HL  
Add DE and previous carry to HL  
Add HL and previous carry to HL  
Add stack pointer and previous carry to HL



| mnemonic |    | assembly/scheme | S Z A V N C | Zilog name | /      | page |
|----------|----|-----------------|-------------|------------|--------|------|
| DAD      | B  | 09              | - - X- 0 X  | ADD        | HL,ss  | 147  |
| DAD      | D  | 19              | - - X- 0 X  | ADD        | HL,ss  | 147  |
| DAD      | H  | 29              | - - X- 0 X  | ADD        | HL,ss  | 147  |
| DAD      | SP | 39              | - - X- 0 X  | ADD        | HL,ss  | 147  |
| DBNZ     | aa | 10,aa-PC        |             | DJNZ       | e      | 235  |
| DCMX     | d  | DD,35,d         | XXXXX1 -    | DEC        | (IX+d) | 129  |
| DCMY     | d  | FD,35,d         | XXXXX1 -    | DEC        | (IY+d) | 129  |
| DCR      | M  | 35              | XXXXX1 -    | DEC        | (HL)   | 129  |
| DCR      | r  | 05V8r           | XXXXX1 -    | DEC        | r      | 129  |
| DCX      | B  | 0B              |             | DEC        | ss     | 160  |
| DCX      | D  | 1B              |             | DEC        | ss     | 160  |
| DCX      | H  | 2B              |             | DEC        | ss     | 160  |
| DCX      | SP | 3B              |             | DEC        | ss     | 160  |
| DCX      | X  | DD,2B           |             | DEC        | IX     | 161  |
| DCX      | Y  | FD,2B           |             | FEC        | IY     | 162  |
| DI       |    | F3              |             | DI         |        | 141  |
| DSC      | B  | ED,42           | XXXXX1 X    | SBC        | HL,ss  | 151  |
| DSC      | D  | ED,52           | XXXXX1 X    | SBC        | HL,ss  | 151  |
| DSC      | H  | ED,62           | XXXXX1 X    | SBC        | HL,ss  | 151  |
| DSC      | SP | ED,72           | XXXXX1 X    | SBC        | HL,ss  | 151  |
| EI       |    | FB              |             | EI         |        | 142  |
| EXX      |    | D9              |             | EXX        |        | 79   |
| HLT      |    | 76              |             | HALT       |        | 140  |
| IM0      |    | ED,46           |             | IM         | 0      | 143  |
| IM1      |    | ED,56           |             | IM         | 1      | 144  |
| IM2      |    | ED,5E           |             | IM         | 2      | 145  |
| IN       | n  | DB,n            |             | IN         | A,(n)  | 253  |
| INMX     | d  | DD,34,d         | XXXXX0 -    | INC        | (IX+d) | 125  |
| INMY     | d  | FD,34,d         | XXXXX0 -    | INC        | (IY+d) | 127  |
| INr      |    | ED,40V8r        | XX0 X0 -    | IN         | r,(C)  | 254  |
| INR      | M  | 34              | XXXXX0 -    | INC        | (HL)   | 124  |
| INR      | r  | 04V8r           | XXXXX0 -    | INC        | r      | 122  |
| INX      | B  | 03              |             | INC        | ss     | 157  |
| INX      | D  | 13              |             | INC        | ss     | 157  |
| INX      | H  | 23              |             | INC        | ss     | 157  |
| INX      | SP | 33              |             | INC        | ss     | 157  |
| INX      | X  | DD,23           |             | INC        | IX     | 158  |

**instruction description**

Add BC to HL

Add DE to HL

Add HL to itself

Add the stack pointer to HL

Decrement B and branch to address aa if B not zero

Decrement (X+d)

Decrement (Y+d)

Decrement (HL)

Decrement register r

Decrement BC

Decrement DE

Decrement HL

Decrement the stack pointer

Decrement X

Decrement Y

Disable the interrupt system

Subtract BC and previous borrow from HL

Subtract DE and previous borrow from HL

Subtract HL and previous borrow from HL

Subtract stack pointer and previous borrow from HL

Enable the interrupt system

Swap BC, DE and HL with BC', DE' and HL'

Halt

Enter interrupt mode 0

Enter interrupt mode 1

Enter interrupt mode 2

Input to A from device n

Increment (X+d)

Increment (Y+d)

Input to register r from address in C

Increment (HL)

Increment register r

Increment BC

Increment DE

Increment HL

Increment the stack pointer

Increment X

| mnemonic |    | assembly/scheme | S Z A V N C | Zilog name | /        | page |
|----------|----|-----------------|-------------|------------|----------|------|
| INX      | Y  | FD,23           |             | INC        | IY       | 159  |
| IN+      |    | ED,AZ           | ? X ? ? 1 - | INI        |          | 256  |
| IN+R     |    | ED,B2           | ? 1 ? ? 1 - | INIR       |          | 258  |
| IN-      |    | ED,AA           | ? X ? ? 1 - | IND        |          | 260  |
| IN-R     |    | ED,BA           | ? 1 ? ? 1 - | INDR       |          | 262  |
| JC       | aa | DA,la,ha        |             | JP         | cc,nn    | 221  |
| JEQ      | aa | CA,la,ha        |             | JP         | cc,nn    | 221  |
| JGE      | aa | D2,la,ha        |             | JP         | cc,nn    | 221  |
| JLS      | aa | DA,la,ha        |             | JP         | cc,nn    | 221  |
| JM       | aa | FA,la,ha        |             | JP         | cc,nn    | 221  |
| JMP      | aa | C3,la,ha        |             | JP         | nn       | 220  |
| JNC      | aa | D2,la,ha        |             | JP         | cc,nn    | 221  |
| JNE      | aa | C2,la,ha        |             | JP         | cc,nn    | 221  |
| JNV      | aa | E2,la,ha        |             | JP         | cc,nn    | 221  |
| JNZ      | aa | C2,la,ha        |             | JP         | cc,nn    | 221  |
| JP       | aa | F2,la,ha        |             | JP         | cc,nn    | 221  |
| JPE      | aa | EA,la,ha        |             | JP         | cc,nn    | 221  |
| JPO      | aa | E2,la,ha        |             | JP         | cc,nn    | 221  |
| JV       | aa | EA,la,ha        |             | JP         | cc,nn    | 221  |
| JZ       | aa | CA,la,ha        |             | JP         | cc,nn    | 221  |
| J<       | aa | FA,la,ha        |             | JP         | cc,nn    | 221  |
| J>=      | aa | F2,la,ha        |             | JP         | cc,nn    | 221  |
| LBCD     | aa | ED,4B,la,ha     |             | LD         | dd,(nn)  | 57   |
| LDA      | aa | 3A,la,ha        |             | LD         | A,(nn)   | 44   |
| LDAX     | B  | 0A              |             | LD         | A,(BC)   | 42   |
| LDAX     | D  | 1A              |             | LD         | A,(DE)   | 43   |
| LDDED    | aa | ED,5B,la,ha     |             | LD         | dd,(nn)  | 57   |
| LHLD     | aa | 2A,la,ha        |             | LD         | HL,(nn)  | 56   |
| LLM      |    | CB,26           | XX0 X0 X    | SLA        | (HL)     | 189  |
| LLMX     | d  | DD,CB,d,26      | XX0 X0 X    | SLA        | (IX+d)   | 189  |
| LLMY     | d  | FD,CB,d,26      | XX0 X0 X    | SLA        | (IY+d)   | 190  |
| LLr      |    | CB,20Vr         | XX0 X0 X    | SLA        | r        | 189  |
| LRM      |    | CB,3E           | XX0 X0 X    | SRL        | (HL)     | 195  |
| LRMX     | d  | DD,CB,d,3E      | XX0 X0 X    | SRL        | (IX+d)   | 195  |
| LRMY     | d  | FD,CB,d,3E      | XX0 X0 X    | SRL        | (IY+d)   | 196  |
| LRr      |    | CB,38Vr         | XX0 X0 X    | SRL        | r        | 195  |
| LrX      | d  | DD,46V8r,d      |             | LD         | r,(IX+d) | 30   |

**instruction description**

Increment Y

Input from address in C, store in (HL), increment HL, decrement B

Same as IN+ above, and repeat if B not zero

Input from address in C, store in (HL), decrement HL and B

Same as in IN- above, and repeat if B not zero

Jump if carry to address aa

Jump if equal to address aa

Jump if greater than or equal to address aa (unsigned)

Jump if less to address aa (unsigned)

Jump if minus to address aa

Jump to address aa unconditionally

Jump if no carry to address aa

Jump if not equal to address aa

Jump if no overflow to address aa

Jump if not zero to address aa

Jump if plus to address aa

Jump if parity even to address aa

Jump if parity odd to address aa

Jump if overflow to address aa

Jump if zero to address aa

Jump if less to address aa (signed)

Jump if greater than or equal to address aa (signed)

Load BC from aa, aa+1

Load A from address aa

Load A from (BC)

Load A from (DE)

Load DE from aa, aa+1

Load HL from aa, aa+1

Logical left shift (HL)

Logical left shift (X+d)

Logical left shift (Y+d)

Logical left shift register r

Logical right shift (HL)

Logical right shift (X+d)

Logical right shift (Y+d)

Logical right shift register r

Load r from (X+d)

| mnemonic |                                | assembly/scheme                    | S Z A V N C | Zilog name | /        | page |
|----------|--------------------------------|------------------------------------|-------------|------------|----------|------|
| LrY      | d                              | FD,46V8r,d                         |             | LD         | r,(IY+d) | 32   |
| LSPD     | aa                             | ED,7B,la,ha                        |             | LD         | dd,(nn)  | 57   |
| LXD      | aa                             | DD,2A,la,ha                        |             | LD         | IX,(nn)  | 59   |
| LXI      | B,nn                           | 01,ln,hn                           |             | LD         | dd,nn    | 53   |
| LXI      | D,nn                           | 11,ln,hn                           |             | LD         | dd,nn    | 53   |
| LXI      | H,nn                           | 21,ln,hn                           |             | LD         | dd,nn    | 53   |
| LXI      | SP,nn                          | 31,ln,hn                           |             | LD         | dd,nn    | 53   |
| LXI      | X,nn                           | DD,21,ln,hn                        |             | LD         | IX,nn    | 54   |
| LXI      | Y,nn                           | FD,21,ln,hn                        |             | LD         | IY,nn    | 55   |
| LYD      | aa                             | FD,2A,la,ha                        |             | LD         | IY,(nn)  | 60   |
| MOV      | M,r                            | 70Vr                               |             | LD         | (HL),r   | 34   |
| MOV      | r,M                            | 46V8r                              |             | LD         | r,(HL)   | 29   |
| MOV      | r <sub>1</sub> ,r <sub>2</sub> | 40V8r <sub>1</sub> Vr <sub>2</sub> |             | LD         | r,r'     | 27   |
| MVI      | M,n                            | 36,n                               |             | LD         | (HL),n   | 39   |
| MVI      | r,n                            | 6V8r,n                             |             | LD         | r,n      | 28   |
| MW+      |                                | ED,A0                              | - - 0 X0 -  | LDI        |          | 83   |
| MW+R     |                                | ED,B0                              | - - 0 0 0 - | LDIR       |          | 85   |
| MW-      |                                | ED,A8                              | - - 0 X0 -  | LDD        |          | 87   |
| MW-R     |                                | ED,B8                              | - - 0 0 0 - | LDDR       |          | 89   |
| NEG      |                                | ED,44                              | XXXXX1 X    | NEG        |          | 135  |
| NOP      |                                | 00                                 |             | NOP        |          | 139  |
| ORA      | M                              | B6                                 | XX0 X0 0    | OR         | (HL)     | 116  |
| ORA      | r                              | B0Vr                               | XX0 X0 0    | OR         | r        | 116  |
| ORI      | n                              | F6,n                               | XX0 X0 0    | OR         | n        | 116  |
| ORX      | d                              | DD,B6,d                            | XX0 X0 0    | OR         | (IX+d)   | 116  |
| ORY      | d                              | FD,B6,d                            | XX0 X0 0    | OR         | (IY+d)   | 116  |
| OTr      |                                | ED,41V8r                           |             | OUT        | (C),r    | 265  |
| OUT      | n                              | D3,n                               |             | OUT        | (n),A    | 264  |
| OUT+     |                                | ED,A3                              | ? X? ? 1 -  | OUTI       |          | 267  |
| OT+R     |                                | ED,B3                              | ? 1 ? ? 1 - | OTIR       |          | 269  |
| OUT-     |                                | ED,AB                              | ? X? ? 1 -  | OUTD       |          | 271  |
| OT-R     |                                | ED,BB                              | ? 1 ? ? 1 - | OTDR       |          | 273  |
| PCHL     |                                | E9                                 |             | JP         | (HL)     | 232  |
| PCX      |                                | DD,E9                              |             | JP         | (IX)     | 233  |
| PCY      |                                | FD,E9                              |             | JP         | (IY)     | 234  |

**instruction description**

Load r from (Y+d)

Load the stack pointer from aa, aa+1

Load X from aa, aa+1

Load BC with immediate data nn

Load DE with immediate data nn

Load HL with immediate data nn

Load the stack pointer with immediate data nn

Load X with immediate data nn

Load Y with immediate data nn

Load Y from aa, aa+1

Store register r at (HL)

Move data from (HL) to register r

Move data from  $r_2$  to  $r_1$

Move immediate data n to (HL)

Move immediate data n to register r

Move word, increment HL and DE, decrement BC, OV on if BC  
not zero

Move word, increment HL and DE, decrement BC, repeat if BC  
not 0

Move word, decrement HL, DE and BC, OV on if BC not 0

Move word, decrement HL, DE and BC, repeat if BC not zero

Negate (two's complement) A

No operation

OR (HL) with A

OR register r with A

OR immediate data n with A

OR (X+d) with A

OR (Y+d) with A

Output register r to address in C

Output A register to device address n

Output from (HL) to address in C, increment HL and decrement B  
Same as OUT+ and repeat if B not zero

Output from (HL) to address in C, decrement HL and B  
Same as OUT- above and repeat if B not zero

Load program counter from HL

Load program counter from X

Load program counter from Y



| mnemonic |     | assembly/scheme | S Z A V N C | Zilog name | /      | page |
|----------|-----|-----------------|-------------|------------|--------|------|
| POP      | B   | C1              |             | POP        | qq     | 72   |
| POP      | D   | D1              |             | POP        | qq     | 72   |
| POP      | H   | E1              |             | POP        | qq     | 72   |
| POP      | PSW | F1              | ??????      | POP        | qq     | 72   |
| POP      | X   | DD,E1           |             | POP        | IX     | 74   |
| POP      | Y   | FD,E1           |             | POP        | IY     | 75   |
| PUSH     | B   | C5              |             | PUSH       | qq     | 69   |
| PUSH     | D   | D5              |             | PUSH       | qq     | 69   |
| PUSH     | H   | E5              |             | PUSH       | qq     | 69   |
| PUSH     | PSW | F5              |             | PUSH       | qq     | 69   |
| PUSH     | X   | DD,E5           |             | PUSH       | IX     | 70   |
| PUSH     | Y   | FD,E5           |             | PUSH       | Y      | 71   |
| RAL      |     | 17              | - - 0 - 0 X | RLCA       |        | 166  |
| RAL\$    |     | CB,17           | XX0 X0 X    | RL         | A      | 180  |
| RAR      |     | 1F              | - - 0 - 0 X | RRA        |        | 170  |
| RAR\$    |     | CB,1F           | XX0 X0 X    | RR         | A      | 186  |
| RC       |     | D8              |             | RET        | cc     | 244  |
| REQ      |     | C8              |             | RET        | cc     | 244  |
| RET      |     | C9              |             | RET        |        | 243  |
| RETI     |     | ED,4D           |             | RETI       |        | 246  |
| RETN     |     | ED,45           |             | RETN       |        | 248  |
| RGE      |     | D0              |             | RET        | cc     | 244  |
| RM       |     | F8              |             | RET        | cc     | 244  |
| RML      |     | CB,16           | XX0 X0 X    | RL         | (HL)   | 180  |
| RMLX     | d   | DD,CB,d,16      | XX0 X0 X    | RL         | (IX+d) | 180  |
| RMLY     | d   | FD,CB,d,16      | XX0 X0 X    | RL         | (IY+d) | 180  |
| RMR      |     | CB,1E           | XX0 X0 X    | RR         | (HL)   | 186  |
| RMRX     | d   | DD,CB,d,1E      | XX0 X0 X    | RR         | (IX+d) | 186  |
| RMRY     | d   | FD,CB,d,1E      | XX0 X0 X    | RR         | (IY+d) | 187  |
| RNC      |     | D0              |             | RET        | cc     | 244  |
| RNE      |     | C0              |             | RET        | cc     | 244  |
| RNV      |     | E0              |             | RET        | cc     | 244  |
| RNZ      |     | C0              |             | RET        | cc     | 244  |
| RP       |     | F0              |             | RET        | cc     | 244  |
| RPE      |     | E8              |             | RET        | cc     | 244  |
| RPO      |     | E0              |             | RET        | cc     | 244  |
| RLC      |     | 07              | - - 0 - 0 X | RLCA       |        | 164  |

**instruction description**

Pops top stack item into BC  
Pops top stack item into DE  
Pops top stack item into HL  
Pops top stack item into AF  
Pops top stack item into X  
Pops top stack item into Y  
Pushes BC onto the stack  
Pushes DE onto the stack  
Pushes HL onto the stack  
Pushes AF onto the stack  
Push X onto the stack  
Push Y onto the stack  
Rotate A left, 9 bits  
Rotate A left, 9 bits (Z80 only)  
Rotate A right, 9 bits  
Rotate A right, 9 bits (Z80 only)  
Return if carry  
Return if equal  
Return unconditionally  
Return from interrupt  
Return from nonmaskable interrupt  
Return if greater than or equal (unsigned)  
Return if minus  
Rotate (HL) left, 9 bits  
Rotate (X+d) left, 9 bits  
Rotate (Y+d) left, 9 bits  
Rotate (HL) right, 9 bits  
Rotate (X+d) right, 9 bits  
Rotate (Y+d) right, 9 bits  
Return if no carry  
Return if not equal  
Return if no overflow  
Return if not zero  
Return if plus  
Return if parity even  
Return if parity odd  
Rotate A left, 8 bits

| mnemonic |     | assembly/scheme | S Z A V N C | Zilog name | /        | page |
|----------|-----|-----------------|-------------|------------|----------|------|
| RLD      |     | ED,6F           | XX0 X0 -    | RLD        |          | 198  |
| RLCM     |     | CB,06           | XX0 X0 X    | RLC        | (HL)     | 174  |
| RLCr     |     | CB,r            | XX0 X0 X    | RLC        | r        | 172  |
| RLCX     | d   | DD,CB,d,06      | XX0 X0 X    | RLC        | (IX+d)   | 176  |
| RLCY     | d   | FD,CB,d,06      | XX0 X0 X    | RLC        | (IY+d)   | 178  |
| RLS      |     | D8              |             | RET        | cc       | 244  |
| RRC      |     | 0F              | - - 0 - 0 X | RRCA       |          | 168  |
| RRCM     |     | CB,0E           | XX0 X0 X    | RRC        | (HL)     | 183  |
| RRCr     |     | CB,8Vr          | XX0 X0 X    | RRC        | r        | 183  |
| RRCX     | d   | DD,CB,d,0E      | XX0 X0 X    | RRC        | (IX+d)   | 183  |
| RRCY     | d   | FD,CB,d,0E      | XX0 X0 X    | RRC        | (IY+d)   | 184  |
| RRD      |     | ED,67           | XX0 X0 -    | RRD        |          | 200  |
| RrL      |     | CB,10Vr         | XX0 X0 X    | RL         | r        | 180  |
| RrR      |     | CB,18Vr         | XX0 X0 X    | RR         | r        | 186  |
| RSC      |     | B7              | XX0 X0 0    | OR         | A        | 116  |
| RST      | t   | C7V8t           |             | RST        | p.       | 250  |
| RV       |     | E8              |             | RET        | cc       | 244  |
| RZ       |     | C8              |             | RET        | cc       | 244  |
| R<       |     | F8              |             | RET        | cc       | 244  |
| R>=      |     | F0              |             | RET        | cc       | 244  |
| SBB      | M   | 9E              | XXXX1 X     | SBC        | A,(HL)   | 112  |
| SBB      | r   | 98Vr            | XXXX1 X     | SBC        | A,r      | 112  |
| SBCD     | aa  | ED,43,la,ha     |             | LD         | (nn),dd  | 62   |
| SBCX     | d   | DD,9E,d         | XXXX1 X     | SBC        | A,(IX+d) | 112  |
| SBCY     | d   | FD,9E,d         | XXXX1 X     | SBC        | A,(IY+d) | 112  |
| SBI      | n   | DE,n            | XXXX1 X     | SBC        | A,n      | 112  |
| SDDED    | aa  | ED,63,la,ha     |             | LD         | (nn),dd  | 62   |
| SHLD     | aa  | 22,la,ha        |             | LD         | (nn),dd  | 61   |
| SIX      | d,n | DD,36,d,n       |             | LD         | (IX+d),n | 40   |
| SIY      | d,n | FD,36,d,n       |             | LD         | IY+d),n  | 41   |
| SPHL     |     | F9              |             | LD         | SP,HL    | 66   |
| SPX      |     | DD,F9           |             | LD         | SP,IX    | 67   |
| SPY      |     | FD,F9           |             | LD         | SP,IY    | 68   |
| SrX      | d   | DD,70Vr,d       |             | LD         | (IX+d),r | 35   |
| SrY      | d   | FD,70Vr,d       |             | LD         | (IY+d),r | 37   |
| SSPD     | aa  | ED,73,la,ha     |             | LD         | (nn),dd  | 62   |
| STA      | aa  | 32,la,ha        |             | LD         | (nn),A   | 47   |

**instruction description**

Rotate left decimal

Rotate (HL) left, 8 bits

Rotate register r left, 8 bits

Rotate (X+d) left, 8 bits

Rotate (Y+d) left, 8 bits

Return if less (unsigned)

Rotate A right, 8 bits

Rotate (HL) right, 8 bits

Rotate register r right, 8 bits

Rotate (X+d) right, 8 bits

Rotate (Y+d) right, 8 bits

Rotate right decimal

Rotate register r left, 9 bits

Rotate register r right, 9 bits

Zero carry bit. Synonym for ORA A

Restart

Return if overflow

Return if zero

Return if less (signed)

Return if greater than or equal (signed)

Subtract (HL) and previous borrow from A

Subtract register r and previous borrow from A

Store BC in aa, aa+1

Subtract (X+d) and previous borrow from A

Subtract (Y+d) and previous borrow from A

Subtract immediate data n and previous borrow from A

Store DE in aa, aa+1

Store HL in aa, aa+1

Store immediate data n at (X+d)

Store immediate data n at (Y+d)

Load the stack pointer from HL

Load the stack pointer from X

Load the stack pointer from Y

Store register r at (X+d)

Store register r at (Y+d)

Store the stack pointer in aa, aa+1

Store A in address aa

| mnemonic |    | assembly/scheme | S Z A V N C | Zilog name | /       | page |
|----------|----|-----------------|-------------|------------|---------|------|
| STAX     | B  | 02              |             | LD         | (BC),A  | 45   |
| STAX     | D  | 12              |             | LD         | (DE),A  | 46   |
| STC      |    | 37              | - - 0 - 0 1 | SCF        |         | 138  |
| SUB      | M  | 96              | XXXX1 X     | SUB        | (HL)    | 110  |
| SUB      | r  | 80Vr            | XXXX1 X     | SUB        | s       | 110  |
| SUBX     | d  | DD,96,d         | XXXX1 X     | SUB        | (IX+d)  | 110  |
| SUBY     | d  | FD,96,d         | XXXX1 X     | SUB        | (IY+d)  | 110  |
| SUI      | n  | D6,n            | XXXX1 X     | SUB        | n       | 110  |
| SYD      | aa | FD,22,la,ha     |             | LD         | (nn),IY | 65   |
| SXD      | aa | DD,22,la,ha     |             | LD         | (nn),IX | 64   |
| TAI      |    | ED,57           | XX0 X0 -    | LD         | A,I     | 48   |
| TAR      |    | ED,5F           | XX0 X0 -    | LD         | A,R     | 49   |
| TIA      |    | ED,47           |             | LD         | I,A     | 50   |
| TRA      |    | ED,4F           |             | LD         | R,A     | 51   |
| TRP      | t  | C7V8t           |             | RST        | p       | 250  |
| XAF      |    | 08              |             | EX         | AF,AF'  | 78   |
| XCHG     |    | EB              |             | EX         | DE,HL   | 77   |
| XRA      | M  | AE              | XX0 X0 0    | XOR        | (HL)    | 118  |
| XRA      | r  | A8Vr            | XX0 X0 0    | XOR        | r       | 118  |
| XRI      | n  | EE,n            | XX0 X0 0    | XOR        | n       | 118  |
| XRX      | d  | DD,AE,d         | XX0 X0 0    | XOR        | (IX+d)  | 118  |
| XRY      | d  | FD,AE,d         | XX0 X0 0    | XOR        | (IY+d)  | 118  |
| XTHL     |    | E3              |             | EX         | (SP),HL | 80   |
| XTX      |    | DD,E3           |             | EX         | (SP),IX | 81   |
| XTY      |    | FD,E3           |             | EX         | (SP),IY | 82   |
| X+BC     |    | DD,09           | - - X- 0 X  | ADD        | IX,pp   | 153  |
| X+DE     |    | DD,19           | - - X- 0 X  | ADD        | IX,pp   | 153  |
| X+SP     |    | DD,39           | - - X- 0 X  | ADD        | IX,pp   | 153  |
| X+X      |    | DD,29           | - - X- 0 X  | ADD        | IX,pp   | 153  |
| Y+BC     |    | FD,09           | - - X- 0 X  | ADD        | IX,rr   | 155  |
| Y+DE     |    | FD,19           | - - X- 0 X  | ADD        | IY,rr   | 155  |
| Y+SP     |    | FD,39           | - - X- 0 X  | ADD        | IY,rr   | 155  |
| Y+Y      |    | FD,29           | - - X- 0 X  | ADD        | IY,rr   | 155  |
| ZAR      |    | AF              | 0 1 0 X0 0  | XOR        | A       | 118  |

**instruction description**

Store A in (BC)

Store A in (DE)

Set carry bit to 1

Subtract (HL) from A

Subtract register r from A, difference in A

Subtract (X+d) from A

Subtract (Y+d) from A

Subtract immediate data n from A

Store Y in aa, aa+1

Store X in aa, aa+1

Transfer IVR to A

Transfer R register to A

Transfer A to IVR

Transfer A to R register

Restart

Swap AF and AF'

Swap contents of HL and DE register pairs

Exclusive OR (HL) with A

Exclusive OR register r with A

Exclusive OR immediate data n with A

Exclusive OR (X+d) with A

Exclusive OR (Y+d) with A

Swap top stack item with HL

Swap top stack item with X

Swap top stack item with Y

Add BC to X

Add DE to X

Add the stack pointer to X

Add X to itself

Add BC to Y

Add DE to Y

Add the stack pointer to Y

Add Y to itself

Zero A register and carry bit. Synonym for XRA A



**SUMMARY OF ASSEMBLER PSEUDO OPERATIONS**

- ORG** Assemble code which follows beginning at location given in operand field of ORG. Anything in ORG operand must have been previously defined.
- END** End of source program.
- TITL** Operand field of TITL will be used as heading for each page of the assembly listing.
- UNL** Suspend listing.
- LST** Resume Listing.
- DATA** Creates one or more single word constants separated by commas. The data types are:  
Decimal — purely numeric field, sign optional;  
Octal — preceded by “at” sign (@);  
Hexadecimal — preceded by colon (:);  
Symbolic — consisting of a defined symbol or sum or difference of symbols and/or numbers whose value can be contained in 8 bits;  
ASCII — a single character enclosed by apostrophes.
- ASC** A string of 36 or fewer characters enclosed by apostrophes.
- DBL** Creates a 16 bit constant equal to the value of the operand field. Data types are the same as for DATA above.
- RES** Reserves a block of memory equal in length to the value of the first operand. The label, if any, attaches to the first word of this block, the one with the lowest memory address. The second operand, separated by a comma may be any 8 bit constant. If the second operand is given the block will be filled with its value by the loader. Anything in the operand of RES must have been previously defined.

- BSS Same as RES above.
- EQU Sets the value of the label to that of the operand. Anything in the operand of an EQU must have been previously defined.
- REM Aligns its own comments field with that of the comment field of an instruction. This allows the comment of the instruction to be continued immediately below, aligned for easy reading. The REM itself does not appear in the assembly listing.



# INDEX

- ACI**, 85
- ADC**, 82
- ADCX**, 82
- ADCY**, 82
- ADI**, 85
- Add/subtract flag, 79
- Address, 28
- AND**, 15,16
- ANX**, 99-101
- ANY**, 99-101
- Arguments, 146
  - addresses in line, 154
  - addresses in register, 150
  - inline, 147
- Arrays, 162
- ASC**, 163
- ASCII codes, 209-210
- Assembler
  - definition of, 33
  - error checks, 324
  - error flags, 36
  - location counter, 322
  - object record types, 325,326
  - symbol table, 321
- Assembly program, 33
  - listing, 34
- Auxiliary Carry Flag, 79
  
- BC**, 66
- BCD**, 182
  - addition, 183-186
  - division, 192-195
  - multiplication, 186-192
  - shifting, 188,189
  - subtraction, 186-188
- BCM**, 263
- BCMx**, 263,264
- BCr**, 263
- BEQ**, 107
- BGE**, 107
  
- Binary addition, 19-22
  - program, 32
  - subtraction, 22
- Binary division, 127-133
- Binary multiplication, 117
  - by special number, 113-116
  - signed, 118-120
- Binary patch, 313-315
- Bit manipulation, 262-264
- Block move instruction, 70-77
- BLS**, 107
- BNC**, 66
- BNE**, 107
- BNZ**, 66
- Borrow, use of, 84
- BRA**, 66,67
- Breakpointing, 307
- BSM**, 263
- BSMY**, 263
- BSr**, 262
- BSS**, 74-76
- BTM**, 262
- BTMX**, 263
- BTr**, 262
- Bubble sort, 174-177
- Busy flag, 200
- BZ**, 66
  
- CALL**, 142
- Calling sequence, 147
- Carry, 19,20
- Carry, use of, 81
- Carry flag, 79
- CC**, 160
- CEQ**, 160
- CGE**, 161
- Checksum, 298
- CLS**, 161
- CM**, 160
- CMA**, 110

**CMP**, 103  
**CMPX**, 103-104  
**CMPY**, 103-104  
**CNC**, 160  
**CNE**, 160  
**CNV**, 160  
**CNZ**, 160  
 Comment field, 35  
 Complementation, 17,18  
 Control function, 197  
 Control port, 197  
 Conversion, 211-234  
     binary to decimal fraction, 231-233  
     binary to decimal integer, 220-230  
     binary to hexadecimal, 234  
     binary to octal, 234  
     decimal to binary integer, 214-219  
**CP**, 160  
**CPE**, 160  
**CPI**, 104  
**CPO**, 160  
**CP+**, 165,166  
**CP+R**, 166-168  
**CV**, 160  
**CZ**, 160  
**C <**, 161  
**C >=**, 161  
  
**DAA**, 183  
**DAC**, 86  
**DAD**, 86  
**DATA**, 37  
 Data port, 197  
 Data transfer, 197  
     input, 286-289  
     output, 289-292  
     under interrupt control, 286-292  
**DBL**, 56,62  
**DBNZ**, 68,69  
**DCMX**, 282  
**DCMY**, 282  
**DCX**, 65  
**DCY**, 65  
 Debug program  
     description, 297-316  
     F function, 302  
  
 I function, 303  
 J function, 315  
 M function, 299-301,315  
 O function, 303  
 R function, 303-306  
 S function, 302  
     / function, 315  
 Device address, 197  
**DI**, 278  
 Divide Fault, 128  
 Dividend, 127,128  
 Displacement  
     of indexed instruction, 60,61  
 Divisor, 127,128  
**DSC**, 87  
  
 Echo input, 206,207  
**EI**, 278,285,286  
 Entry point, 143  
**EQU**, 39,61  
 Exclusive OR, 16,17  
**EXX**, 131  
  
 FIFO Stack, 73  
 Fixed point, 90  
 Flag bits, 30  
 Flags, 78-79  
 Floating point  
     addition, 238-241  
     bias, 236  
     conversion, 247-249  
     exponent, 235,236  
     fraction, 235,236  
     noisy mode, 245  
     normalized, 236  
     overflow, 241  
     precision loss, 243-245  
     sign, 235,236  
     true hexadecimal exponent, 246  
     underflow, 241  
     unnormalized, 236  
 Frequency plot, 250  
  
 Half add, 20  
 Hertzprung-Russell diagram, 264-275  
 Hexadecimal, 2,11  
 Histogram, 250

Image region, 257  
 Immediate mode, 63  
**IM0**, 279  
**IM1**, 279  
**IM2**, 280  
 Inclusive OR, 13  
 Index register, 60,61  
**INR**, 65  
 Instruction, 29  
     execution cycle, 29  
     field, 34  
     register, 29  
 Interface, 196  
 Interrupt Vector Register, 280  
 Interrupts, 276-296  
     arming, 278  
     disablement, 278  
     disarming, 278-279  
     enablement, 278  
     sequence of events, 277  
**INX**, 65  
**INY**, 65  
  
**JC**, 68  
**JEQ**, 107  
**JGE**, 107  
**JLS**, 107  
**JM**, 68  
**JMP**, 70  
**JNC**, 68  
**JNE**, 107  
**JNV**, 68  
**JNZ**, 68  
**JP**, 68  
**JPE**, 70  
**JPO**, 70  
**JV**, 68  
**JZ**, 68  
**J <**, 107  
**J > =**, 107  
  
 Label field, 35  
     rules for, 40  
**LBCD**, 55  
**LDA**, 54,55  
**LDAX**, 59  
  
**LDED**, 55  
**LHLD**, 55  
**LLM**, 93  
**LLMX**, 93  
**LLMY**, 93  
 Load, 54  
 Logical difference, 16  
 Logical product, 15  
 Logical sum, 15  
 Looping, 67,68  
**LRM**, 93  
**LRMX**, 93  
**LRMY**, 93  
**LrX**, 60  
**LrY**, 60  
**LSPD**, 135  
**LXD**, 55  
**LXI**, 63  
**LYD**, 55  
  
 Masking, 16  
**MOV**, 57,58  
 Move, 54  
 Multiple precision, 81  
     addition, 81-84,87  
     division, 134  
     multiplication, 133  
     negation, 111  
     shifts, 94-98  
     subtraction, 84,88  
**MVI**, 63  
**MW+R**, 70  
  
**NEG**, 110  
 Negation, 22-26  
 Nonmaskable Interrupt, 293  
  
 Object program, 34  
 Octal, 10  
 One's complement, 18  
 Operand field, 35  
**ORA**, 98,99  
**ORG**, 36  
**ORX**, 99-101  
**ORY**, 99-101  
 Overflow, 24-26  
 Overflow/parity flag, 79



**PCHL**, 141  
**PCX**, 141  
**PCY**, 141  
 Pointer, 56,57  
 Polling, 280  
**POP**, 137  
 Powers of 2, 8  
 Program counter, 29  
 Pseudo operation, 37  
 Pseudo registers, 303-306  
**PUSH**, 136  
  
 Quotient, 128  
  
 Radix deflation method, 247-249  
**RALS**, 92  
**RARS**, 92  
**RC**, 160  
 Real Time Clock, 282  
 Real Time executive, 292,293  
 Reentrancy, 294  
 Registers, 30,31  
**REM**, 50  
 Remainder, 128  
**REQ**, 160  
**RES**, 74-76  
 Restart instruction, 279  
     effective address, 279  
**RET**, 142  
**RETI**, 293  
**RETN**, 293  
**RGE**, 161  
**RLC**, 91  
**RLCM**, 92  
**RLD**, 188,189  
**RLS**, 161  
**RM**, 160  
**RNC**, 160  
**RNE**, 160  
**RNV**, 160  
**RNZ**, 160  
**RP**, 160  
**RPE**, 160  
**RPO**, 160  
**RRC**, 91  
**RRCM**, 92

**RRCX**, 92  
**RRD**, 188,189  
**RrL**, 92  
**RrR**, 92  
**RSC**, 50  
**RST**, 296  
**RV**, 160  
**RZ**, 160  
**R <**, 161  
**R > =**, 161  
  
**SBCD**, 55  
**SBI**, 85  
 Scale factor, 89  
     in multiplication, 127  
**SDED**, 55  
 Sense function, 197  
 Shifting, 18,19  
**SHLD**, 55  
 Sign flag, 78  
**SIX**, 282  
**SIY**, 282  
 Source program, 34  
     buffer, 41  
**SrX**, 60  
**SrY**, 60  
**SSPD**, 135  
**STA**, 54,55  
 Stack pointer, 31,135  
**STAX**, 59  
 Store, 54  
 Subroutine, 143  
**SUI**, 85  
**SXD**, 55  
**SYD**, 55

Tables, 162  
**TAI**, 280  
**TCA**, 111  
 Text editor, 41-53  
**TIA**, 280  
**TITL**, 53  
**TRP**, 296  
 Two's complement system, 22-26

vanity check, 212  
Vectored Interrupt, 281

Word, 27  
length, 28

**XAF**, 138  
**XRA**, 98,99  
**XRX**, 99-101  
**XRY**, 99-101  
**XTHL**, 140  
**XTX**, 140  
**XTY**, 140  
**X+SP**, 140  
**X+X**, 121  
**X-Y** plot, 257-261

**Y+SP**, 140

Zero flag, 78



To receive object code of the assembler and debug programs . . .  
Fill out both parts of the coupon on the facing page and send it to:  
Northern Technology Books  
Box 62  
Evanston, IL 60204

Please print or type all information. The lower part of the coupon will be used as your mailing label.

Disclaimer

Though the cassettes and paper tapes are prepared under stringent quality control, the great variability of physical parameters such as read head alignment makes it impossible to guarantee that they will be readable on any particular computer system. The cassettes and paper tapes are therefore supplied with no warranty, express or implied, as to their suitability for use with any particular computer system, and the user so agrees by the return of the coupon.

While the assembler and debug programs and all examples in the text have been prepared with all prudent caution and care and have been subjected to extensive testing under diverse conditions, neither they nor any other computer program can be certified to be completely error free. Northern Technology Books assumes no liability for either the function of the programs or any direct or indirect damage which may occur as a consequence of their use, and the user of the programs so agrees by the purchase of this book.

